UNIVERSITY OF CALIFORNIA, SAN DIEGO

Divide and Conquer in Parallel Complexity and Proof Theory

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Mathematics

by

Stephen Austin Bloch

Committee in charge:

Professor Samuel R. Buss, Chair
Professor Russell Impagliazzo
Professor Alfred Manaster
Professor Christos Papadimitriou
Professor Jeff Remmel

1992

The dissertation of Stephen Bloch is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

1992

TABLE OF CONTENTS

LIST OF FIGURE

ACKNOWLEDGMENTS

A work like this usually depends on many different kinds of support, including financial, emotional, professional, and intellectual. This dissertation was born at the 1991 Conference on Proof Theory, Complexity, and Arithmetic in Prague, Czechoslovakia, which I attended with financial support from NSF/ČSAV grant INT-8914569, thanks to Sam Buss and Gaisi Takeuti; later stages of the writing were supported financially by grants from the University of Manitoba, thanks to Judy Goldsmith.

I owe two people for the essential job of hounding me to actually finish the thing in finite time: my mother Dorothy Beling, and Judy Goldsmith. In addition, thanks are due to Sam Buss, Judy Goldsmith, Al Manaster, and Jeff Remmel for their invaluable professional advice on how to be a mathematician.

Sam Buss, Patrick Dymond, Russell Impagliazzo, Christos Papadimitriou, Jeff Remmel, and Victor Vianu taught me most of what I know, and more that I've forgotten, about computational complexity and mathematical logic. Part I of the dissertation was inspired by Stephen Cook's talk at the 1991 conference, presenting his work with Stephen Bellantoni; the author has subsequently benefited from discussions with Stephen Bellantoni and Daniel Leivant on this work. Part II was inspired by informal discussions with Richard Kaye at the same conference. Other helpful suggestions on mathematical content came from Judy Goldsmith and Andrew Klapper. And of course, thanks are due to my advisor, Sam Buss, without whose advice, corrections, and frequent insistence that "you can prove a stronger result than that" this dissertation would not exist.

VITA

27 January, 1964	Born, Lexington, Kentucky
1983-1985	Student Co-op, MITRE Corporation
1986	B.S, Department of Mathematics, Virginia Tech (also fulfilled requirements for B.S. in Computer Science and minor in Music)
1986	Research Assistant in Logic Programming for Dr. John Roach, Department of Computer Science, Virginia Tech
1986–1991	Teaching Assistant, Department of Mathematics University of California, San Diego
1989	M.S., Department of Mathematics, University of California, San Diego
1992	Visiting Research Assistant in Computational Complexity for Dr. Judy Goldsmith, Department of Computer Science, University of Manitoba
1992	Doctor of Philosophy, Department of Mathematics, University of California, San Diego

PUBLICATIONS

"Functional Characterizations of Uniform Log-depth and Polylog-depth Circuit Families." Proceedings of 1992 IEEE Conference on Structure in Complexity Theory.

FIELDS OF STUDY

Major Field: Mathematics

Computational Complexity Theory.
Professors Sam Buss, Patrick Dymond, Russell Impagliazzo, Christos Papadimitriou, Michael Saks, and Victor Vianu

Mathematical Logic.
Professors Sam Buss and Jeff Remmel

ABSTRACT OF THE DISSERTATION

Divide and Conquer in Parallel Complexity
and Proof Theory

by

Stephen Austin Bloch
Doctor of Philosophy in Mathematics
University of California, San Diego, 1992
Professor Samuel R. Buss, Chair

This dissertation comprises two largely independent parts. Part I deals with the classes of functions computable by suitably uniform, multi-output, bounded-fanin Boolean circuit families of depths (a) a constant times the log of the input length (henceforth "log-depth"), or (b) a polynomial in the log of the input length (henceforth "polylog-depth"). The corresponding relation classes are known in the literature as ALOGTIME (or U_{E^*}-uniform NC^1) and alternating polylog time (or polylog deterministic space) respectively. The latter contains U_{E^*}-uniform NC, but appears to differ from it in allowing 2^{polylog} rather than merely polynomial circuit size. We adapt a technique due to Leivant (who called it "tiered recursion") and Bellantoni and Cook (who called it "safe recursion") by modifying the recursion scheme to a divide-and-conquer, or branching, recursion, producing as a result the cleanest characterization to date of the functions computable by polylog-depth circuits. We introduce an additional variation, "very safe recursion", which allows us to elegantly characterize the functions computable by log-depth circuits, and which shows potential to characterize linear time and other important classes.

Part II deals with weak subtheories of Peano arithmetic collectively referred to as "theories of bounded arithmetic", in particular the theories R^i_k, whose induction scheme is analogous to the divide-and-conquer recursion used in Part I and which

seems a natural theory in which to formalize statements about parallel computation. Richard Kaye showed that, for various syntactic formula classes Φ (e.g. Σ_i), the subtheory of Peano arithmetic with induction on Φ formulæ is Φ-conservative over the subtheory with induction only on *parameter-free* Φ formulæ, those containing no free variables except the induction variable. We extend his work in two important ways: first, we prove similar conservation statements for theories axiomatized with prefix induction or divide-and-conquer induction rather than the standard successor induction; and second, we present our proofs in a proof-theoretic, rather than a model-theoretic, framework. The latter difference allows us to prove not only logical conservation but stronger proof-theoretic statements of computational interest.

Chapter 1

Introduction

The first four sections of this chapter are an introduction to the field and methods of computational complexity. Section 1.4, in particular, discusses circuit complexity and uniformity, and points out a few ways the treatment in this dissertation differs from standard approaches in the field. Section 1.5 summarizes our results. The reader familiar with the field may either skip directly to this section, or read Section 1.4 first for the few nonstandard definitions used in Section 1.5.

1.1 Computational Complexity

The field of computational complexity treats computation with limited resources such as time, memory, and processing power. We might say that the whole field stems from the original question "Can Problem X be solved with resources Y?" To make sense of this question, researchers in the field adopt the convention that a *problem* is a question with infinitely many instances — typically a question containing a place-holder that may be filled in by infinitely many different sets of input data. A *solution* to a problem is a single, uniform algorithm which correctly solves all (or, in probabilistic models, most) instances of the problem. And we define a *resource bound* as not a fixed amount of memory, or a fixed number of clock cycles, but rather a function indicating, for each set of input data, a corresponding ensemble of resources;

the feature of interest is then how rapidly the resource requirements for the "best" solution grow with the size (or some other natural measure) of the input.

Unfortunately, there are many problems of practical interest for which we do not knowingly possess a "best" algorithm. For example, even so familiar a task as multiplication is only known to be insoluble in fewer than $O(n)$ primitive computational steps, while the best known algorithms take $O(n \log(n) \log(\log(n)))$. (Here n denotes the total length, in bits, of the two factors to be multiplied, and thus the information content of the input.) Even if we ask not for a particular algorithm but only for an assurance that an algorithm of certain resource bounds *exists*, we almost never know with confidence that a given resource bound is necessary and sufficient.

If we cannot know for certain how hard a given problem is, we can still gain practical information from the *relative* difficulty of different problems. Results of this sort typically reduce one problem to another, implying not only that the former's innate difficulty is "not much harder" than the latter, but that *any* algorithm for the latter, even if not itself optimal, can be easily converted into an almost equally efficient algorithm for the former. Conversely, if we want evidence that a problem is intractable (e.g. breaking a cryptographic system) we may proceed by showing it is *at least as difficult* as some other problem widely considered intractable. Complexity theorists often generalize these approaches, defining a complexity class as the set of *all* problems solvable within such-and-such resource bounds, and showing reducibility, subset, and equality relationships among such classes.

In answering such questions, we frequently find it useful to view a particular complexity class in very different, but provably equivalent, ways. For example, under certain circumstances a time-bounded general digital computer, an inductively-defined class of functions, a syntactically-defined class of logical formulæ, a class of graphs defined by connectivity and path length, and an axiomatization of number theory may all in a sense encompass the same computational limitations. In fact, current research uses all these models heavily, and this dissertation deals with several of them.

1.2 A Brief History of Computational Complexity

Researchers of the early and mid-20th century like Gödel, Church, Turing, Post, and Kleene developed various models of formal computation and showed that each model computed the same class of problems. This single, robust class of "recursive" problems gave rise to a rich theory of computability and incomputability. It was soon recognized, however, that a problem that takes a billion years to solve may be *formally* but not *feasibly* computable. Intuitively, a problem is feasibly computable if some computer that might actually be constructed in the next hundred years or so could solve realistic-sized instances of it rapidly enough that the answer would still be useful to the person who posed the instance of the problem. Increasing numbers of researchers then began to try to approximate this notion with formal definitions of "feasible".

In the 1960's a developing consensus defined feasible problems as those whose innate time requirements grew polynomially in the lengths[1] of their inputs; this definition, too, was insensitive to a wide variety of changes in computational model, and was named $\mathcal{P}$ (for yes-or-no decision problems) or $\mathcal{FP}$ (for functions). But $\mathcal{P}$ was not nearly as well-behaved as was the class of recursive problems, and it was often difficult to tell whether a given problem had a polynomial-time algorithm. In particular, a wide variety of practical problems involved searching an exponential-sized space for any object with certain polynomially-verifiable properties.

Such problems could clearly be solved in exponential time, yet they intuitively seemed a more restricted class: they *could* be solved by machine in polynomial time if only the machine could periodically ask an oracle which half of the space might

[1]The length $|x|$ of a parameter x is defined as the number of bits necessary to represent it. When we think of our inputs as bit-strings, this is simply the length of the string; when we think of our inputs as natural numbers, it is $\lceil \log(x+1) \rceil$, the number of bits in the binary representation of x with no leading zeroes; it differs by at most one from $\log_2(x)$, and so when additive constants are irrelevant we shall ignore the distinction, writing for example $||x||$ for $\log(|x|)$.

contain a suitable object. More precisely, a decision problem is in "nondeterministic $\mathcal{P}$", or $\mathcal{NP}$, if there is an oracle-querying, polynomial-time machine such that for all instances of the problem, the correct answer is "yes" iff there is a sequence of oracle answers that leads the machine to answer "yes". Cook [Coo71] described this class of problems, and showed one $\mathcal{NP}$ problem, the satisfiability of Boolean formulæ with free propositional variables, "complete" in the sense that if it were in $\mathcal{P}$, they all would be, thus framing the most famous question in theoretical computer science, $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$. Karp [Kar72] showed several more apparently dissimilar problems to be likewise $\mathcal{NP}$-complete, and the study of such problems, and the structure of $\mathcal{NP}$ and related classes, became something of a cottage industry [GJ79]. One such related class is the "co-$\mathcal{NP}$" problems, those that replace the existential question "does there exist an object in the search space with such-and-such properties?" by the universal question "do all objects in the search space have such-and-such properties?". Since $\mathcal{P}$ is closed under negation, a problem is co-$\mathcal{NP}$ iff its negation is $\mathcal{NP}$. Meyer and Stockmeyer [MS72], [Sto76] extended these notions: where $\mathcal{NP}$ (respectively, co-$\mathcal{NP}$) problems asked about the existence (nonexistence) of objects with certain $\mathcal{P}$ properties, $\mathcal{P}^{\mathcal{NP}}$ could be defined as the decision problems polynomial-time verifiable with an oracle for $\mathcal{NP}$ (or equivalently for co-$\mathcal{NP}$), $\mathcal{NP}^{\mathcal{NP}}$ (co-$\mathcal{NP}^{\mathcal{NP}}$) could be defined as asking about the existence (nonexistence) of objects with $\mathcal{P}^{\mathcal{NP}}$ properties, and so on. The collection of all these classes is called the *polynomial time hierarchy* or PH.

A further extension, by Chandra, Kozen, and Stockmeyer [CKS81], defined *alternating Turing machines*, which could make either existential or universal guesses at will within the course of a computation, and in particular could alternate between the two polynomially often rather than only constantly many times as was the case in PH. Another interpretation of machines with existential or universal guesses views each such guess as a parallel branching point, at which the machine sets two submachines to work simultaneously, each assuming a different answer to the guess, and computes the OR (for existential guesses) or the AND (for universal guesses) of their conclusions[2]. Technological advances in the 1970's and 1980's gave this parallel interpretation practical significance, as it began to be feasible to build machines with large numbers of parallel processors. But implementing alternating polynomial-time algorithms appears to require exponentially many processors, still beyond the ability of current technology, so more work has been done on *speeding up* problems already known to be in $\mathcal{P}$ by means of parallelism: interest has therefore grown in alternating log-time and alternating polylog-time algorithms.

1.3 Characterizing Complexity Classes

The original definitions of complexity classes typically depend heavily on Turing machines, circuits, or some other model explicitly constructed to describe computation. As scientists, we would feel more confident that our work described a universal phenomenon, something inherent in the world rather than computer design as the accidents of history have brought it about, if we had equivalent descriptions not depending on a machine definition. For an analogy from logic, while mathematicians are convinced by Gödel's construction of formally unprovable statements of, say, Peano arithmetic, the result is somewhat unsatisfying because the statements in question incorporate a notion of proof and are explicitly constructed to be unprovable. The Paris-Harrington-Ramsey theorem, by contrast, can be stated in purely combinatorial terms, requires no diagonalization, and contains *no* notion of proof, and thus more satisfyingly demonstrates the incompleteness of Peano arithmetic.

One approach to machine-independent characterization of a complexity class is to define a set of functions inductively so that it comprises exactly the functions in the class, but its inductive definition is mathematically justifiable independently of any particular model of computation. This is in fact the most natural way to

[2]Alternating Turing machines, by their nature, solve decision problems rather than computing functions. To discuss function computation in the ATM setting, the convention is to require the ATM to compute a function's *bit-graph*, i.e. given values $\vec{x}$ and i, to answer whether the i-th bit of $f(\vec{x})$ is 1 or 0.

describe certain high-level classes such as the primitive recursive and the partial recursive functions, but until recently such recursive-functions characterizations of practical computational complexity classes have depended on rather artificial base functions, unnatural restrictions on recursion and other constructors, or large numbers of construction schemes. Bellantoni and Cook [BC92] developed the cleanest characterization to date of $\mathcal{FP}$, the functions computable in polynomial time, replacing a number of unnatural bounds and base functions by the relatively simple notion of a two-sorted universe, and Bellantoni [Bel92a] extends the technique to characterize functions in the Meyer-Stockmeyer polynomial hierarchy. Part I of the present work extends the technique in two other ways to give the cleanest characterizations to date of functions computable by uniform log-depth and polylog-depth circuit families (see Section 1.4 for definitions). Leivant and Marion [LM92] point out that all four of these characterizations rely on presenting the natural numbers in various ways as a free algebra, generalize the technique to other free algebras, and point out that in certain cases the only base functions necessary are the constructors of the free algebra.

Several other, related, types of machine-independent characterization come from mathematical logic. Classical theorems due to Post, Kleene, and Matijasevič characterize the recursive, r.e, co-r.e, etc. relations by counting the syntactic alternation of unbounded quantifiers in first-order formulæ defining those relations. By analogy, if unbounded quantifiers are disallowed and we count the alternation of *bounded* quantifiers in a suitable language, we can syntactically characterize the relations at any desired level of the polynomial hierarchy (see [Bus86], chapter 1). A more philosophically interesting characterization, the main theorem of [Bus86], shows the *functions* at any desired level of the polynomial hierarchy to be exactly those which are not only definable by formulæ of a corresponding syntactic complexity, but also provably total in a correspondingly restricted axiomatization of number theory. This work has led to a number of similar characterizations of complexity classes in terms of the proof strength of such restricted theories (see e.g. [All91], [Clo89a], [BKT92], [Ara92]), and thence to considerable interest in the differences in proof strength among the theories themselves. Part II of the present work investigates several such axiomatizations. Such logical characterizations have also appeared in second-order form (e.g. [Kra90], [Tak92], [KT90], [Lei91], [Lei92], and the second half of [Bus86]). In a different approach, many researchers consider the totality and other properties of a function not as a single statement in a first- or second-order theory but rather as an infinite family of statements in a quantifier-free theory; we can then draw conclusions of computational interest from how fast the proofs of those statements grow (see e.g. [Coo75], [Bus87], [Clo92], and [BIK$^+$92]).

1.4 Circuit Complexity

The three most common models of parallel computation are parallel random-access machines (PRAMs), alternating Turing machines, and circuit families. We shall not discuss PRAMs in this paper, and have already mentioned alternating Turing machines. A formal circuit is a finite, directed, acyclic graph whose source and sink nodes correspond to the inputs and outputs, respectively, of a physical circuit, and whose non-source nodes are each labelled with a Boolean function such as AND, OR, or NOT; each node computes a Boolean value in the natural way from the Boolean values of its inputs. Since each circuit is finite, a problem with infinitely many instances can only be solved by an infinite *family* of circuits, one for each possible number n of input bits. We can then define the complexity class NC^k, for $k = 1, 2, \ldots$, to be the set of problems solvable by single-output circuit families in which each gate has at most two inputs, the depth (longest path from input to output) is $O(\log^k(n))$ and the size (total number of nodes) is polynomial in n. The related classes AC^k eliminate the restriction to two-input gates and allow the fanin of gates to vary with n. The class $NC = \cup_k NC^k$ comprises those problems solvable by polynomial-size, polylog-depth, bounded-fanin circuit families. It is trivial that $NC^k \subseteq AC^k$ for all k, and not difficult to show that $AC^k \subseteq NC^{k+1}$, and so $AC = \cup_k AC^k = NC$. A good introduction to this area is [Coo85].

Most treatments of circuit complexity concentrate on decision problems, and assume that each circuit has only one output gate; such circuits can be coaxed to compute functions by the "bit-graph" method described above. For example, decision problems solvable by NC^0 circuit families are considered trivial, since they have constant depth, constant fanin, and one output, and hence the output can only depend on constantly many bits of the input. In this paper we consider circuits that compute functions to be *multi-output*, and define the class FNC^k to be the set of *functions* computable by multi-output circuit families of constant fanin, polynomial size, and $\log(n)^k$ depth. In contrast to the single-output case, the function class FNC^0 contains many useful functions such as the bitwise AND of two equal-length bit-strings, which can clearly be computed by constant-depth, bounded-fanin circuits. This additional power comes at a price: we must keep track of output size as well as total circuit size. A single-output circuit may have size exponential in its depth; we can preserve this property in our setting by allowing output size to be at most exponential in depth.

Regardless of output size, the above definition of circuit computation is unsatisfactory in a way, because there are families of very simple circuits that "solve" nonrecursive problems like whether Turing machine n halts when given a blank input tape. The paradox can be avoided by requiring the family to be *uniform*; that is, there must be a computationally simple (not to mention recursive!) way, given n, to construct the circuit for n inputs. The late 1970's and early 1980's produced many definitions of uniformity, of which several are still in common use today — polynomial-time uniformity, log-space uniformity, the FO uniformity of [Imm89], and the U_D, U_E, and U_{E^*} uniformities of [Ruz81].

Polynomial-time and log-space uniformity are the easiest to state, requiring simply that there exist a polynomial-time (respectively, log-space) Turing machine which on input n constructs a complete description of the circuit for n inputs. Polynomial-time is the least stringent uniformity criterion in common use, but is sufficient to prove the classical result that $\mathcal{P}$ is precisely the set of problems solvable by uniform, polynomial-sized circuits. Since NC is polynomial size with a further restriction to polylog depth, it follows immediately that, under any of the uniformity criteria discussed in this paper, NC is within $\mathcal{P}$. The converse is an important open question analogous to $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$; even so apparently small a class as U_E-uniform NC^1 is not known to be properly contained in $\mathcal{P}$.

Obviously, a polynomial-sized circuit cannot be constructed in less than polynomial time. Ruzzo, in [Ruz81], classified the various notions of uniformity and suggested that, rather than requiring a uniformity condition to *construct* an entire circuit, one should merely require the uniformity condition to *recognize local properties* of such a circuit; this made sublinear-time[3] uniformity criteria useful. Ruzzo distinguished between recognizing a circuit family's "direct connection language" — intuitively, the type of any specified gate and whether one specified gate was a child of another — and its "extended connection language", in which there might be queries about not only children but descendants along an arbitrary path. He then defined several notions of uniformity and showed a number of inclusions and equalities among them, of which the most relevant to this paper are the following:

Definition 1 (Ruzzo) *The* direct connection language *of a circuit family is the set of 4-tuples $\langle n, g, p, y \rangle$ such that n is the binary representation of the number of inputs to the circuit, g is a binary number identifying a gate in the circuit, $p \in \{\epsilon, L, R\}$, and either $p = \epsilon$ and y is the type of gate g in some fixed encoding, or $p \in \{L, R\}$ and y is a binary number identifying the left or right, respectively, input gate to g.*

Definition 2 (Ruzzo) *The* extended connection language *of a circuit family of size $Z(n)$ is the set of 4-tuples $\langle n, g, p, y \rangle$ such that n is the binary representation of the number of inputs to the circuit, g is a binary number identifying a gate in the circuit, $p \in \{L, R\}^*$, and either $p = \epsilon$ and y is the type of gate g in some fixed encoding,*

[3] Under the strictest definition of a Turing machine, sublinear time is meaningless. The usual convention in this area gives Turing machines a form of random access: one work tape distinguished as the "index tape", and the ability to, at any time, read the symbol on the input tape at the position represented in binary by the contents of the index tape.

or p represents a nontrivial path of at most $\log(Z(n))$ left/right decisions and y is a binary number identifying the descendant of g reached by following that path.

Definition 3 (Ruzzo) *A family of bounded-fanin circuits of size $Z(n)$ and depth $T(n)$ is U_D-uniform if its direct connection language is recognizable in deterministic time $O(\log(Z(n)))$.*

Definition 4 (Ruzzo) *A family of bounded-fanin circuits of size $Z(n)$ and depth $T(n)$ is U_E-uniform if its* extended *connection language is recognizable in deterministic time $O(\log(Z(n)))$.*

Definition 5 (Ruzzo) *A family of bounded-fanin circuits of size $Z(n)$ and depth $T(n)$ is said to be U_{E^*}-uniform if its extended connection language is recognizable by an* alternating *Turing machine in time $O(T(n))$ and space $O(\log(Z(n)))$.*

Clearly U_E-uniformity is at least as stringent a criterion as either U_D- or U_{E^*}-uniformity, in the sense that any U_E-uniform circuit family is also U_D-uniform and, if $T(n) = \Omega(\log(Z(n)))$, U_{E^*}-uniform. (For the single-output circuits of Ruzzo's setting, the assumption $T(n) = \Omega(\log(Z(n)))$ is vacuous if the circuit is to depend on all the bits of its input. Since $Z(n) = \Omega(n)$ for such circuits, counting the input gates in the size, it follows that $T(n) = \Omega(\log(n))$ is also vacuous for Ruzzo. In our multi-output setting we still have $Z(n) = \Omega(n)$, but neither of the other assumptions necessarily holds.) It turns out, however, not much more stringent, as the following two theorems show:

Theorem 1 (Ruzzo) *If $T(n) = \Omega(\log(n))$ and $Z(n) = n^{\Omega(1)}$, and a language $\mathcal{L}$ is accepted by a U_D-uniform circuit family of size $Z(n)$ and depth $T(n)$, then $\mathcal{L}$ is accepted by a (possibly different) U_E-uniform circuit family of size $Z(n)^{O(1)}$ and depth $O(\max(T(n), \log(Z(n)) \log\log(Z(n))))$.*

Theorem 2 (Ruzzo) *If $T(n) = \Omega(\log(n))$ and $Z(n) = n^{\Omega(1)}$, and a language $\mathcal{L}$ is accepted by a U_{E^*}-uniform circuit family of size $Z(n)$ and depth $T(n)$, then $\mathcal{L}$ is accepted by a U_E-uniform circuit family of size $Z(n)^{O(1)}$ and depth $O(T(n))$.*

Thus if we are interested in whether a given problem has circuits of a given depth (at least log) and size (at least polynomial), and if we ignore constant factors of depth and polynomial size blowups, then U_E-uniformity and U_{E^*}-uniformity are equivalent; if we also ignore depth factors of $\log\log(Z(n))$, U_D-uniformity is equivalent to the other two.

Theorem 3 (Ruzzo) *If $T(n)$, $Z(n)$, and $\mathcal{L}$ are as in Theorem 2, then $\mathcal{L}$ is accepted by an alternating Turing machine in space $\log(Z(n))$ and time $T(n)$.*

Theorem 4 (Ruzzo) *If $T(n) = \Omega(\log(n))$ and $Z(n) = n^{\Omega(1)}$, and a language $\mathcal{L}$ is accepted by an alternating Turing machine in space $\log(Z(n))$ and time $T(n)$, then $\mathcal{L}$ is accepted by a U_E-uniform (hence U_{E^*}-uniform) circuit family of size $Z(n)^{O(1)}$ and depth $T(n)$.*

Thus if we are interested in the complexity of a given problem, alternating Turing machine time and space correspond to U_{E^*}-uniform circuit size and depth respectively, to within a polynomial in the former and a constant factor in the latter. In particular, the class of problems solvable in alternating time and space $O(\log(n))$ (henceforth ALOGTIME) comprises exactly the problems solvable with NC^1 circuits whose extended connection language is in ALOGTIME. Thus we might call U_{E^*}-uniform NC^1 "self-uniform", in that its uniformity condition is no stronger than itself; this argues for its naturalness as a complexity class.

A different notion of uniformity is usually used in order to study classes below NC^1. Immerman [Imm87], [Imm89] defined a variety of complexity classes purely in terms of expressibility in first-order logic.

Definition 6 (Immerman, mostly) *Let $\mathcal{C}$ be a class of predicates on natural numbers, and let $\tau_{\mathcal{C}}$ be a language containing $=,\leq$, and finitely many other predicate symbols corresponding to predicates in $\mathcal{C}$. A problem $P(x)$ is in $FO(\mathcal{C})$ if there is a closed, first-order formula θ in the language $\tau_{\mathcal{C}}$ such that $P(x)$ holds if and only if θ is true in the finite model $\{0, 1, \ldots |x| - 1\}$, with $\mathcal{C}$ predicates interpreted in the natural way (which may depend on x).*

The class FO is an abbreviation for $FO(BIT, Z)$, where $BIT(i, j)$ is interpreted as the bit of j with significance 2^i, and $Z(i)$ is interpreted as $BIT(i, x)$.

The class FO, defined in terms of first-order formulæ with random access over finite models, is a machine-independent description of a form of uniform AC^0: as [BIS90] shows, FO problems are computable by constant-depth, unbounded-fanin circuit families whose *direct* connection languages are recognizable in deterministic log time. If we extend $\mathcal{C}$ with other predicates, the resulting class comprises problems solvable by FO-uniform constant-depth circuits containing gates computing those other predicates in addition to unbounded-fanin AND and OR gates. This has become the predominant approach for studying classes between AC^0 and NC^1.

Since we are eventually interested in characterizing NC^1 and related classes, we want a uniformity criterion based on recognizing extended, not direct, connection languages; yet in parts of the proofs we shall be concerned with constant-depth circuit families, for which U_{E^*} makes no sense (it would call for an alternating machine with log space, but only constant time) and U_E is apparently too strict. So at the risk of reopening a tired subject, we shall use yet another notion of uniformity.

Definition 17: *A family of bounded-fanin circuits of size $Z(n)$ and depth $T(n)$ is said to be $U_{E'}$-uniform if its extended connection language is recognizable by an alternating Turing machine with time bound $O(\log(Z(n)))$.*

This definition allows us to construct useful constant-depth circuits, in the multi-output setting, but is not *too* dissimilar from standard notions of uniformity:

Theorem 6: *Let a circuit family $\mathcal{F}$ have depth $T(n)$ and size $Z(n)$.*
If $T(n) = O(\log(Z(n)))$ and $\mathcal{F}$ is U_{E^*}-uniform, then $\mathcal{F}$ is $U_{E'}$-uniform.
Conversely, if $T(n) = \Omega(\log(Z(n)))$ and $\mathcal{F}$ is $U_{E'}$-uniform, then $\mathcal{F}$ is U_{E^*}-uniform.

Corollary 7: *If $Z(n) = n^{\Omega(1)}$ and $T(n)$ is both $\Omega(\log(n))$ and $O(\log(Z(n)))$, then the following are equivalent:*

- *Language $\mathcal{L}$ has U_E-uniform circuits of size $Z(n)^{\Theta(1)}$ and depth $\Theta(T(n))$,*
- *Language $\mathcal{L}$ has U_{E^*}-uniform circuits of size $Z(n)^{\Theta(1)}$ and depth $\Theta(T(n))$,*

- *Language $\mathcal{L}$ has $U_{E'}$-uniform circuits of size $Z(n)^{\Theta(1)}$ and depth $\Theta(T(n))$.*

1.5 Scope of the Present Work

A functional characterization of a computational resource bound is a class of functions defined inductively by composition, recursion, and similar operations on a few base functions, comprising exactly the functions computable within the resource bounds. Part I of this dissertation defines two such classes and shows them to be the functions computable by uniform, bounded-fanin circuit families of depth $O(\log(n))$ and $\log(n)^{O(1)}$ respectively (with output length exponential in the depth), or equivalently by alternating Turing machines with time bounds $O(\log(n))$ and $\log(n)^{O(1)}$ respectively. These functional characterizations are an improvement in elegance over previous characterizations of the same computational classes, as our function classes contain no explicit mention of growth rates, and as they require only very simple base functions and only two operations on functions — forms of composition and recursion.

The functions discussed in Part I operate on a two-sorted universe of bit-strings in which each function may take both "safe" and "normal" parameters. Following [BC92] and [Lei90], we notate such functions in the form $f(\vec{x};\vec{y})$, where the parameters appearing before the semicolon are normal and those after the semicolon are safe. We define our function class by closing a finite set $BASE$ under the operations of "safe composition" and "safe [or very safe] divide-and-conquer recursion". (The following definitions are presented here without comment; they appear again, with discussion and examples, in Chapter 2.)

Definition 7: *Let $BASE$ denote the following set of functions:*
$\pi_k^{i,j}(x_1,\ldots,x_i;x_{i+1},\ldots,x_{i+j}) = x_k$, *the projection functions (for $1 \leq k \leq i+j$),*

$$\begin{array}{rcl} \lambda, 0, 1 & & \textit{constants} \\ MSP(;x,y) & = & \textit{the leftmost } |x| \dot{-} |y| \textit{ bits of } x \textit{ (the "most significant part"} \\ & & \textit{of } x\textit{)} \end{array}$$

$$
\begin{aligned}
LSP(;x,y) &= \textit{the rightmost } |y| \textit{ bits of } x\textit{, with } |y| \dot{-} |x| \textit{ leading zeroes} \\
& \textit{(the "least significant part" of } x\textit{)} \\
Cond(;b,u,v) &= \begin{cases} LSP(;u,\max(u,v)) & \textit{if } b\textit{'s low bit is 1} \\ LSP(;v,\max(u,v)) & \textit{if } b=\lambda \textit{ or its low bit is 0} \end{cases} \\
Conc(;x,y) &= \textit{the concatenation of } x \textit{ and } y \\
Bh(;x) &= \textit{the rightmost } \lceil |x|/2 \rceil \textit{ bits of } x \textit{ (the "back half" of } x\textit{)} \\
Fh(;x) &= \textit{the leftmost } \lfloor |x|/2 \rfloor \textit{ bits of } x \textit{ (the "front half" of } x\textit{)} \\
Not(;x) &= \textit{the one's complement of } x \\
Or(;x,y) &= \textit{the bitwise OR of } x \textit{ and } y\textit{, first padding the shorter of} \\
& \textit{the two with leading zeroes up to the length of the longer} \\
Ins_0(;x) &= x \textit{ with a 0 inserted after each bit} \\
Ins_1(;x) &= x \textit{ with a 1 inserted after each bit}
\end{aligned}
$$

Definition 8 (Bellantoni & Cook) : *We say function $f(\vec{x};\vec{y})$ is defined by* safe composition *from functions $g(x_1 \ldots x_r; y_1 \ldots y_s)$, $u_1(\vec{x};) \ldots u_r(\vec{x};)$, and $v_1(\vec{x};\vec{y}) \ldots v_s(\vec{x};\vec{y})$ if*

$$f(\vec{x};\vec{y}) = g(u_1(\vec{x};) \ldots u_r(\vec{x};); v_1(\vec{x};\vec{y}) \ldots v_s(\vec{x};\vec{y})$$

Definition 11: *A function $f(z,b,\vec{x};\vec{y})$ is said to be defined by* safe divide-and-conquer recursion *(henceforth* safe DCR*) from functions $h(z,\vec{x};\vec{y},u_1,u_2)$ and $g(z,\vec{x};\vec{y})$ if*

$$
\begin{aligned}
&f(z,b,\vec{x};\vec{y}) = \\
&\quad = \begin{cases} g(z,\vec{x};\vec{y}) & \textit{if } |z| \leq \max(|b|,1) \\ h(z,\vec{x};\vec{y}, f(Fh(;z),b,\vec{x};\vec{y}), f(Bh(;z),b,\vec{x};\vec{y})) & \textit{if } |z| > \max(|b|,1) \end{cases}
\end{aligned}
$$

Definition 12: *A function $f(z,b,\vec{x};\vec{y})$ is said to be defined by* very safe divide-and-conquer recursion *(henceforth* very safe DCR*) from functions $h(;z,\vec{x},\vec{y},u_1,u_2)$ and*

$g(z, \vec{x}; \vec{y})$ *if*

$$f(z, b, \vec{x}; \vec{y}) =$$
$$= \begin{cases} g(z, \vec{x}; \vec{y}) & \textit{if } |z| \leq \max(|b|, 1) \\ h(; z, \vec{x}, \vec{y}, f(Fh(; z), b, \vec{x}; \vec{y}), f(Bh(; z), b, \vec{x}; \vec{y})) & \textit{if } |z| > \max(|b|, 1) \end{cases}$$

In Chapters 3 and 4 we study the classes of functions $vsc(BASE)$ and $sc(BASE)$ definable from BASE by finitely many applications of safe composition and very safe (respectively, safe) divide-and-conquer recursion. The main theorems are

Theorem 9: *The following are equivalent:*

(1) $f(\vec{x};) \in vsc(BASE)$

(2) $f(\vec{x};)$ *has* $U_{E'}$*-uniform, log-depth, polynomial-size circuits*

(3) the language $\{(i, b, \vec{x}) | Bit(i, f(\vec{x};)) = b\}$ *is recognized by an alternating* $O(\log)$*-time machine, and* f *has polynomial growth rate.*

Theorem 19: *The following are equivalent:*

(1) $f(\vec{x};) \in sc(BASE)$

(2) $f(\vec{x})$ *has* $U_{E'}$*-uniform, polylog-depth circuits, and* f *has* $2^{poly(||\vec{x}||)}$ *growth rate*

(3) the i-th bit of $f(\vec{x})$ *can be computed from* i *and* $\vec{x}$ *in alternating polylog time, and* f *has* $2^{poly(||\vec{x}||)}$ *growth rate*

(4) $f(\vec{x})$ *can be computed by a deterministic Turing machine in polylogarithmic space (not counting a write-only output tape)*

Part II discusses theories of bounded arithmetic, which are weak subtheories of Peano arithmetic formed by restricting the induction rule in various ways. We present a general framework for showing conservation results between theories with different forms of induction, then use it to give conservation results between a number of such theories and their "parameter-free" analogues, which are formed by restricting induction to apply only when there are no free variables except the variable of induction.

Let $\mathcal{L}_1$ be the language with constant symbols 0 and 1, relation symbols $=$ and $\leq$, and operator symbols for addition, subtraction, multiplication, floored division, and the "length" function $|x|$. For $k > 1$, the language $\mathcal{L}_k$ is defined by adding operator symbols for the "smash" functions $\#_2, \#_3, \ldots \#_k$ defined as follows:

$$\begin{aligned} x\#_1 y &= 2^{|x|+|y|} - 1 \\ x\#_2 y &= 2^{|x|\cdot|y|} - 1 \\ x\#_3 y &= 2^{|x|\#_2|y|} - 1 \\ &\vdots \\ x\#_{i+1} y &= 2^{|x|\#_i|y|} - 1 \\ &\vdots \end{aligned}$$

The formula classes $\Sigma^b_{i,k}$ and $\Pi^b_{i,k}$ are a generalization of the classes Σ^b_i and Π^b_i defined in [Bus86]. As usual in systems of first-order logic, formulæ are constructed by applying relations to terms and by applying quantifiers and Boolean connectives to smaller formulæ. A quantifier appearing in formulæ may be restricted by attaching explicit bounds to the variable, as in $(\forall x \leq t)\phi(x)$ where t is a term; such a quantifier is called a *bounded quantifier*. A further special case is the class of *sharply bounded quantifiers*, those bounded quantifiers in which the bounding term t is of the form $|u|$, for u another term.

Definition 37: *The classes $\Sigma^b_{i,k}$ and $\Pi^b_{i,k}$ are the smallest sets of formulæ satisfying the following inductive definition.*

- *$\Sigma^b_{0,k}$ and $\Pi^b_{0,k}$ are both the set of formulæ in language $\mathcal{L}_k$ in which all quantifiers are sharply bounded.*

- $\Pi^b_{i,k} \subseteq \Sigma^b_{i+1,k}$.

- $\Sigma^b_{i,k} \subseteq \Pi^b_{i+1,k}$.
- *If* $A \in \Sigma^b_{i+1,k}$ *and* t *is a term in language* $\mathcal{L}_k$*, then the formulæ* $(\exists x \leq t)A$ *and* $(\forall x \leq |t|)A$ *are also in* $\Sigma^b_{i+1,k}$.
- *If* $A \in \Pi^b_{i+1,k}$ *and* t *is a term in language* $\mathcal{L}_k$*, then the formulæ* $(\forall x \leq t)A$ *and* $(\exists x \leq |t|)A$ *are also in* $\Pi^b_{i+1,k}$.
- *If* $A, B \in \Sigma^b_{i,k}$ *(respectively* $\Pi^b_{i,k}$*), then so are* $A \wedge B$ *and* $A \vee B$.
- *If* $A \in \Sigma^b_{i,k}$ *(respectively* $\Pi^b_{i,k}$*), then* $\neg A \in \Pi^b_{i,k}$ *(respectively* $\Sigma^b_{i,k}$*).*

The theories T^i_k, S^i_k, and R^i_k are defined from a finite set of axioms describing how the symbols of the language relate to one another and allowing progressively weaker types of induction called IND, $PIND$, and DCI respectively, restricted to $\Sigma^b_{i,k}$ formulæ $A(x, \vec{y})$.

IND:

$$A(0, \vec{y}) \wedge (\forall x)(A(x, \vec{y}) \supset A(x+1, \vec{y})) \supset (\forall x)A(x, \vec{y})$$

PIND:

$$A(0, \vec{y}) \wedge (\forall x) \left(A\left(\left\lfloor \frac{x}{2} \right\rfloor, \vec{y} \right) \supset A(x, \vec{y}) \right) \supset (\forall x)A(x, \vec{y})$$

DCI:

$$A(0, \vec{y}) \wedge A(1, \vec{y}) \wedge (\forall x)(A(Fh(x), \vec{y}) \wedge A(Bh(x), \vec{y}) \supset A(x, \vec{y})) \supset (\forall x)A(x, \vec{y})$$

where Fh and Bh are essentially the "front half" and "back half" functions in Definition 7. (This is something of an oversimplification: in Part II we actually redefine Fh and Bh slightly, with the result that DCI needs not only 0 and 1 but also 2 and 3 as base cases.)

The main results of Part II demonstrate that for each of the systems T^i_k and S^i_k for $i, k \geq 1$, and R^i_k for $i \geq 1, k \geq 2$, the usual formulation of the system is $\forall\exists\Sigma^b_i$-conservative over the system with only parameter-free induction. The results (and some of the proof techniques) resemble Theorem 5.5 of [Kay87], which shows conservation between parameter-free and ordinary forms of $I\Sigma_n$, IE_n, and $I\exists_n$. (E_n is similar to $\Sigma^b_{n,1}$, but counts sharply-bounded quantifiers as no different from bounded ones. Σ_n is analogous to $\Sigma^b_{n,1}$ but counting the alternation of *unbounded* quantifiers and ignoring *bounded* ones. $\exists_n$ is the same thing, but counting bounded quantifiers as no different from unbounded ones. Thus IE_n is similar to, and a subtheory of, T^n_1.) Our results differ from Kaye's in three ways: they apply to theories with other than the classical IND induction, our proofs are proof-theoretic rather than model-theoretic, and as a result we can prove not only logical conservation but stronger proof-theoretic statements such as "if θ has a proof in T^i_k that contains no non-$\Sigma^b_{i,k}$ formulæ, then θ has a proof in *parameter-free* T^i_k, still containing no non-Σ^b_i formulæ." (See Section 6.8 for definitions).

To get the proofs to go through for the systems R^i_k we need a few extra functions that are easily definable in R^1_2, but not apparently in its parameter-free analogue. We could get around this by adding them to the language $\mathcal{L}_1$, but as the parameter-free systems can define their graphs and prove all their necessary properties except totality, we take the more frugal approach of adding totality axioms.

Part I

Functional Characterizations

Chapter 2

Preliminaries to Part I

2.1 Introduction

Much of the history of the study of computational complexity has been informed by Cobham's elegant recursion-theoretic characterization in [Cob65] of the polynomial-time functions as the closure of certain simple functions under composition and a form of recursion. Specifically, Cobham's "bounded recursion on notation" iterates a number of times equal not to the integer value of some parameter x, but rather to $|x|$, the length of x when written in binary notation and also requires that the function computed by such recursion grow no faster than some pre-existing function, i.e. no faster than some function defined by composition of base functions alone. This bounding requirement effectively obliges anyone using bounded recursion on notation to accompany every use thereof with a proof that the function so defined has only polynomial growth rate.

Various researchers over the years have developed other formal characterizations of complexity classes, most relying on some sort of explicit bound to keep growth rates under control. For example, Buss [Bus86] presented a slightly different version of Cobham's characterization, then used it to *logically* characterize $\mathcal{FP}$ and the functions at each level of the polynomial hierarchy as the provably total functions in a corresponding weak form of Peano arithmetic called Bounded Arithmetic; this theory did however require a language expressly designed to produce polynomial growth rates. Leivant [Lei91], [Lei92] gave the first characterization of $\mathcal{FP}$ with *no* reference to polynomial growth, by a form of second-order arithmetic with comprehension for positive quantifier-free formulæ. The second-order framework corresponded to the scheme of "tiered recursion" in [Lei90], which adds finer control to such recursion-theoretic characterizations by making the underlying universe many-sorted.

Bellantoni and Cook, in [BC92], independently developed a comparable "tiering" technique after reading [Lei91], and gave a concrete recursion-theoretic characterization of polynomial-time functions with no explicit mention of polynomials. This characterization looks similar to Cobham's original characterization, but omits the bounding requirement and instead distinguishes syntactically between "normal" function parameters and "safe" ones, on which a restricted set of operations are permitted. Most importantly, recursion is permitted only on normal parameters, while each step of that recursion must treat the results of previous steps as safe. Thus if a function f is defined by iterating function h, the time and space bounds on h depend only on the arguments to f, not on the rest of the recursive definition in which it is used. As a result, the size bounds on functions hold automatically rather than being a separate and inconvenient stipulation in the recursion scheme.

Other researchers have used similar techniques to Cobham's and/or Buss's to describe space-bounded and parallel complexity classes. Lind, in [Lin74], gave a recursion-theoretic characterization of logspace, and Allen [All91] gave logical and recursion-theoretic characterizations of NC, both (like Cobham) explicitly bounding the growth rates of functions defined in their systems. Compton and LaFlamme [CL90] present recursion-theoretic and logical characterizations of uniform NC^1, relying on a polynomial size bound implicit in their finite-models approach. Clote, in [Clo89b], introduced a scheme of *concatenation recursion on notation* that corresponds to computing each bit of a function in parallel, and characterized uniform NC^1, AC^0, and a variety of other parallel classes by combining concatenation recursion on notation, log-depth bounded recursion, and a somewhat unnatural tree-evaluation function. A more elegant characterization, in [Clo92], replaces the tree-evaluation function by Cobham-style recursion on notation with a *constant* (rather than polynomial) bound, relying on Barrington's work [Bar89] on bounded-width branching programs; both of these Clote papers, like [Bus86], rely on a language expressly designed to have polynomial growth rate. Arai [Ara92] characterizes uniform NC^1 with systems of bounded arithmetic based on the "upward tree recursion" technique of [CL90], and Clote and Takeuti [CT92] do the same with the induction scheme we call LLIND; both Arai and Clote-Takeuti use a language with only linear growth rate but explicitly specify $O(\log(n))$ recursion depth (and hence $2^{O(\log(n))} = poly(n)$ size).

By combining Bellantoni & Cook's "safe parameters" technique with Allen's "divide and conquer" recursion, we have avoided both explicit size/depth bounds and unnatural base functions and characterized alternating polylog time (or uniform polylog-depth circuits, or polylog space). The additional notion of "very safe" recursion introduced here converts the above characterization into one of alternating log time (ALOGTIME, or uniform NC^1), still without explicit size or depth bounds.

The rest of this chapter defines the terminology, recursion schemes, and functions used in the proofs. Chapter 3 states and proves the characterization of NC^1; most of this chapter is concerned with simulating the computation of a machine within our function class. Chapter 4 states and proves the similar characterization of functions computable by uniform $\log^{O(1)}$-depth circuits. Finally, Chapter 5 gives an alternate, and perhaps more natural, formulation of the two characterizations, compares our methods with previous ones, and discusses directions for further research.

2.2 Definitions of the Function Algebras

We shall think of our data not as natural numbers but primarily as finite strings over the alphabet $\Sigma = \{0, 1\}$. Such strings will however often be interpreted as binary numbers, treating the rightmost symbol as least significant; by convention, the rightmost symbol of a string will be called bit 0, the second-rightmost bit 1, and so on. We use the symbol λ to denote the empty string.

We define, not within the system but purely for notational convenience, the function $Bit(i, x)$ which yields the bit in position i of x (i.e. the bit with significance 2^i).

We now define a set of base functions. Following [BC92], we shall distinguish between "normal" and "safe" parameters of a function by writing the function with all normal parameters first, separated by a semicolon from the safe parameters, e.g. $f(x, y; z)$.

Definition 7 *Let BASE denote the following set of functions:*
$\pi_k^{i,j}(x_1, \ldots, x_i; x_{i+1}, \ldots, x_{i+j}) = x_k$, *the projection functions,*

$$
\begin{array}{rcl}
\lambda, 0, 1 & & \textit{constants} \\
MSP(;x,y) & = & \textit{the leftmost } |x| \dot{-} |y| \textit{ bits of } x \textit{ (the "most significant part"} \\
& & \textit{of } x\textit{)} \\
LSP(;x,y) & = & \textit{the rightmost } |y| \textit{ bits of } x\textit{, with } |y| \dot{-} |x| \textit{ leading zeroes} \\
& & \textit{(the "least significant part" of } x\textit{)} \\
Cond(;b,u,v) & = & \begin{cases} LSP(;u, \max(u,v)) & \textit{if } b\textit{'s low bit is 1} \\ LSP(;v, \max(u,v)) & \textit{if } b = \lambda \textit{ or its low bit is 0} \end{cases} \\
Conc(;x,y) & = & \textit{the concatenation of } x \textit{ and } y \\
Bh(;x) & = & \textit{the rightmost } \lceil |x|/2 \rceil \textit{ bits of } x \textit{ (the "back half" of } x\textit{)} \\
Fh(;x) & = & \textit{the leftmost } \lfloor |x|/2 \rfloor \textit{ bits of } x \textit{ (the "front half" of } x\textit{)} \\
Not(;x) & = & \textit{the one's complement of } x \\
Or(;x,y) & = & \textit{the bitwise OR of } x \textit{ and } y\textit{, first padding the shorter of} \\
& & \textit{the two with leading zeroes up to the length of the longer} \\
Ins_0(;x) & = & x \textit{ with a 0 inserted after each bit} \\
Ins_1(;x) & = & x \textit{ with a 1 inserted after each bit}
\end{array}
$$

All these functions are reasonably familiar, except perhaps the last two.

We illustrate their operation with an example: $Ins_0(;0110) = 00101000$. Of course, only one of the two functions is needed: if Ins_1 were omitted, it could be defined as $Not(;Ins_0(;Not(;x)))$. These functions may be thought of as dual in a sense to the $Concat$,Fh, and Bh functions: where the latter treat a $2n$-bit string as the concatenation of two n-bit strings, the Ins_0 and Ins_1 functions treat a $2n$-bit string as the concatenation of n 2-bit strings, or equivalently as the interleaving of two n-bit strings. Allen, among others, has commented on the apparent necessity of such dual views for characterizing parallel complexity classes, and it appears again in the *Copies* and *Explode* functions of Chapter 6. Leivant and Marion [LM92] claim most of the above base functions, including Ins_0 and Ins_1, can be eliminated, but the author has not seen detailed proofs.

Note that each BASE function is defined in such a way that the length of the output is a simple function of the lengths of the inputs, to avoid worries about circuit uniformity. (This is the reason for the LSP's in the definition of $Cond$: the answer u or v is padded to a uniform length that will hold either.)

Definition 8 (Bellantoni & Cook) *We say function $f(\vec{x};\vec{y})$ is defined by* safe composition *from functions $g(x_1 \ldots x_r; y_1 \ldots y_s)$, $u_1(\vec{x};) \ldots u_r(\vec{x};)$, and $v_1(\vec{x};\vec{y}) \ldots v_s(\vec{x};\vec{y})$ if*

$$f(\vec{x};\vec{y}) = g(u_1(\vec{x};) \ldots u_r(\vec{x};); v_1(\vec{x};\vec{y}) \ldots v_s(\vec{x};\vec{y})$$

Note that, according to this definition, a safe parameter to f can never be used as a normal parameter to any of the u's or v's, nor can it affect the normal parameters of g. However, a normal parameter to f can be passed into a safe position by using a projection function for one of the v's. This "trap-door" effect will also be preserved by our recursion schemes.

Definition 9 (Bellantoni & Cook) *A function $f(z,\vec{x};\vec{y})$ is defined by* safe recursion on notation *from functions $g(\vec{x};\vec{y})$, $h_0(z,\vec{x};u,\vec{y})$, and $h_1(z,\vec{x};u,\vec{y})$ if*

$$\begin{aligned} f(0,\vec{x};\vec{y}) &= g(\vec{x};\vec{y}) \\ f(s_i(z),\vec{x};\vec{y}) &= h_i(z,\vec{x};f(z,\vec{x};\vec{y}),\vec{y}) \quad \textit{for } s_i(z) \neq 0 \end{aligned}$$

where $s_i(z) = 2z + i$, *for* $i = 0, 1$.

When a function accepts a parameter as "safe" it effectively promises not to use that parameter to control the depth of a recursion. (So what is "safe" from the caller's point of view might better be called "sensitive" from that of the called function.) All else being equal, then, a function which can make that promise is more powerful than one which cannot. For example, the concatenation function could be defined as $Conc(x, y;)$, $Conc(x; y)$, or $Conc(; x, y)$. Although all three compute exactly the same value, the third is the most flexible, and choosing it over the others to include in $BASE$ will prove crucial in showing size and time bounds.

The major difference between the present system and that of [BC92] is the replacement of their scheme of "safe recursion on notation" by various forms of "safe divide-and-conquer recursion". Divide-and-conquer recursion (splitting a large datum into two pieces of roughly equal size, and recursing on both) is familiar to every computer scientist, but to the author's knowledge its first use to characterize a complexity class functionally was in [All91], where the following scheme, which Allen calls "polynomial size branching recursion", is used to build recursion-theoretic characterizations of logspace and NC. (A cosmetically different recursion scheme was called "limited recursion" in [Bus86] and "upward tree recursion" in [CL90].)

Definition 10 (Allen, modified for our notation) *A function* $f(y, \vec{x})$ *is defined by* polynomial size branching recursion *from functions* g *and* h *with polynomial size bound* $p(n)$ *if*

$$f(y, \vec{x}) \;=\; \begin{cases} g(y, \vec{x}) & \text{if } |y| \leq 1 \\ h(y, \vec{x}, f(Fh(; y), \vec{x}), f(Bh(; y), \vec{x})) & \text{if } |y| > 1 \end{cases}$$

and for all $a, \vec{b}$,

$$|f(a, \vec{b})| \;\leq\; p\left(|a| + \sum |\vec{b}|\right)$$

We shall make similar definitions, but by distinguishing between normal and safe parameters we can dispense with the explicit size bound p.

Definition 11 *A function $f(z, b, \vec{x}; \vec{y})$ is said to be defined by* safe divide-and-conquer recursion *(henceforth* safe DCR*) from functions $h(z, \vec{x}; \vec{y}, u_1, u_2)$ and $g(z, \vec{x}; \vec{y})$ if*

$$f(z, b, \vec{x}; \vec{y}) =$$
$$= \begin{cases} g(z, \vec{x}; \vec{y}) & \text{if } |z| \leq \max(|b|, 1) \\ h(z, \vec{x}; \vec{y}, f(Fh(; z), b, \vec{x}; \vec{y}), f(Bh(; z), b, \vec{x}; \vec{y})) & \text{if } |z| > \max(|b|, 1) \end{cases}$$

Note: One difficulty in "programming" with DCR is that each leaf of the binary computation tree computes the same function of (for the most part) the same data, distinguished only by the portion of z each is given. If the recursion were required to proceed until $|z| \leq 1$, a more obvious definition, there could be only constantly many (i.e. $|\Sigma| + 1$) distinct leaf values in the whole tree, and we might expect safe DCR to define different classes of functions depending on the size of the alphabet. Specifying b, and hence when the recursion is to bottom out, allows leaves to be distinguished by larger data structures than a single bit, which appears necessary in order to prove the main theorems. We maximize $|b|$ with 1 because if $|b| = 0$, the recursion goes into an infinite loop by repeatedly finding $Bh(; 1) = 1$.

Example: Using safe DCR, we can define a function $COPY(x; y)$ which concatenates $|x|$ copies of y (and hence has the same growth rate as the "smash" operator $\#$, defined by $x\#y = 2^{|x|\cdot|y|} - 1$).

$$COPY(x; y) = \begin{cases} \lambda & \text{if } |x| = 0 \\ y & \text{if } |x| = 1 \\ Conc(; COPY(Fh(; x); y), COPY(Bh(; x); y)) & \text{if } |x| > 1 \end{cases}$$

or more formally,

$$COPY(x; y) = aux(x, 1; y)$$
$$aux(x, b; y) = \begin{cases} g(x; y) & \text{if } |x| \leq \max(|b|, 1) \\ Conc(; aux(Fh(; x), b; y), & \text{if } |x| > \max(|b|, 1) \\ \qquad aux(Bh(; x), b; y)) & \end{cases}$$
$$g(x; y) = Cond(; Fh(; Conc(; 1, x)), y, \lambda)$$

The $Fh(;Conc(;1,x))$ in the definition of g allows us to distinguish between the cases $|x| = 0$ and $|x| = 1$. Aside from this, the formal definition is obvious and unilluminating; henceforth we shall usually give such definitions by DCR in the first, less formal manner.

Applying safe DCR to this in turn yields the following:

$$\#_3(x, y;) \quad = \quad \begin{cases} 1 & \text{if } |x| = 0 \\ y & \text{if } |x| = 1 \\ COPY(y; \#_3(Fh(;x), y;)) & \text{if } |x| > 1 \end{cases}$$

This function has the property $|\#_3(x, y)| = |y|^{||x||}$: the equation is clearly true for $|x| \leq 1$, and each time $|x|$ doubles, the length of $\#_3(x, y)$ is multiplied by $|y|$, for a total length $|y|^{||x||}$ (this equation holds even for $|x|$ not a power of 2). This gives the above function the growth rate $2^{||x|| \cdot (||y|| + O(1))}$, effectively the same as that of the $\#_3$ operator defined by $x \#_3 y = 2^{|x| \# |y|}$.

Definition 12 *A function* $f(z, b, \vec{x}; \vec{y})$ *is defined by* very safe DCR *from functions* $h(; z, \vec{x}, \vec{y}, u_1, u_2)$ *and* $g(z, \vec{x}; \vec{y})$ *if*

$$f(z, b, \vec{x}; \vec{y}) =$$
$$= \begin{cases} g(z, \vec{x}; \vec{y}) & \textit{if } |z| \leq \max(|b|, 1) \\ h(; z, \vec{x}, \vec{y}, f(Fh(;z), b, \vec{x}; \vec{y}), f(Bh(;z), b, \vec{x}; \vec{y})) & \textit{if } |z| > \max(|b|, 1) \end{cases}$$

Note: This differs from safe DCR in that safe DCR allows the iterated function h to itself be defined by iteration, so long as said iteration does not depend on the results of previous calls to h; in very safe DCR, the iterated function takes no normal parameters whatsoever, and thus cannot be defined by iteration at all. For example, the above definition of $COPY$ is actually very safe, but the definition of $\#_3$ is not (in fact, we prove in Lemma 10 that $\#_3$ *cannot* be defined by very safe DCR).

Definition 13 *$sc(BASE)$, the "safe closure of BASE", denotes the closure of the BASE functions under both safe composition and safe DCR.*

Definition 14 *$vsc(BASE)$, the "very safe closure of BASE", denotes the closure of the BASE functions under both safe composition and very safe DCR.*

It is often useful to be able to combine a constant number of values into a tuple. If all the values $\vec{x}$ are guaranteed to be the same length, this can be easily done. Let Z be any string of the same length as each of the x's (we'll ignore its value anyway, so we could actually use one of the x's), then define a family of combining and extracting functions as follows:

Definition 15

$COMB_1(;x) = x$

$COMB_{2i}(;x_1, \ldots x_{2i}) =$

$\quad = Conc(;COMB_i(;x_1, \ldots x_i), COMB_i(;x_{i+1}, \ldots x_{2i}))$ *for* $i > 0$

$COMB_{2i+1}(;x_1, \ldots x_{2i+1}) =$

$\quad = Conc(;COMB_{i+1}(;x_1, \ldots x_{i+1}), COMB_{i+1}(;x_{i+2}, \ldots x_{2i+1}, Z))$ *for* $i > 0$

$EXTR_1^1(;y) = y$

$$EXTR_k^i(;y) = \begin{cases} EXTR^i_{\lceil k/2 \rceil}(;Fh(;y)) & \textit{if } i \le \lceil k/2 \rceil \\ EXTR^{i-\lceil k/2 \rceil}_{\lceil k/2 \rceil}(;Bh(;y)) & \textit{if } i > \lceil k/2 \rceil \end{cases}$$

Thus $EXTR_k^i(;COMB_k(;a_1 \ldots a_k)) = a_i$ for all $1 \le i \le k$.

Definition 16 *A collection of functions $f_1(z, b, \vec{x}; \vec{y}), \ldots f_k(z, b, \vec{x}; \vec{y})$ is said to be defined by* simultaneous, safe divide-and-conquer recursion *from functions $h_1, \ldots h_k$ and $g_1, \ldots g_k$ if for all $i \le k$,*

$$f_i(z, b, \vec{x}; \vec{y}) = $$
$$= \begin{cases} g_i(z, \vec{x}; \vec{y}) & \textit{if } |z| \le \max(|b|, 1) \\ h_i(z, \vec{x}; \vec{y}, f_1(Fh(;z), b, \vec{x}; \vec{y}) \ldots f_k(Fh(;z), b, \vec{x}; \vec{y}), & \textit{if } |z| > \max(|b|, 1) \\ \qquad f_1(Bh(;z), b, \vec{x}; \vec{y}) \ldots f_k(Bh(;z), b, \vec{x}; \vec{y})) & \end{cases}$$

The definition of simultaneous, very safe divide-and-conquer recursion *is identical, except that all parameters to the h_i's are treated as safe.*

Lemma 5 *If for all* $b, \vec{x}, \vec{y}, i, j$ *we have* $|g_i(z, \vec{x}; \vec{y})| = |g_j(z, \vec{x}; \vec{y})|$ *for* $|z| \leq |b|$, *and if the equations* $(|u_1| = \ldots = |u_k|) \wedge (|v_1| = \ldots = |v_k|)$ *imply* $|h_i(z, \vec{x}; \vec{y}, \vec{u}, \vec{v})| = |h_j(z, \vec{x}; \vec{y}, \vec{u}, \vec{v})|$, *and if* $g_1 \ldots g_k, h_1 \ldots h_k \in sc(BASE)$, *and if functions* $\vec{f}$ *are defined by simultaneous, safe DCR from* $\vec{g}$ *and* $\vec{h}$, *then each of the functions* $\vec{f}$ *is in* $sc(BASE)$. *That is, under these equal-length conditions, simultaneous safe DCR can be simulated by ordinary safe DCR.*

A similar lemma holds for $vsc(BASE)$.

Note: The equal-length hypothesis could be weakened considerably, but this form of the lemma is easy to prove, and suffices for our purposes.

Proof for either $sc(BASE)$ or $vsc(BASE)$: The hypotheses of equal length, by an obvious induction, imply $|f_i(z, b, \vec{x}; \vec{y})| = |f_j(z, b, \vec{x}; \vec{y})|$ for all $z, b, \vec{x}, \vec{y}, i, j$. Using the tuple-coding functions $COMB$ and $EXTR$, we can define a single function $F(z, b, \vec{x}; \vec{y})$ as the encoding of the tuple $f_1(z, b, \vec{x}; \vec{y}), \ldots f_k(z, b, \vec{x}; \vec{y})$. That is, given the value of a recursive call to F, we extract each f_i with $EXTR_k^i$, then apply the appropriate h's and combine the results with $COMB_k$. ∎

2.3 Conventions on Circuits

Many of the results herein concern circuit families. Our circuits may contain gates computing any Boolean function with at most two inputs, most importantly the constants 0 and 1, identity, negation, two-input AND, and two-input OR. Circuits that compute functions (as opposed to relations) are assumed to be multi-output unless stated otherwise. All circuit families in this paper are uniform in the following sense, which may be called $U_{E'}$:

Definition 17 *A family of bounded-fanin circuits of size* $Z(n)$ *and depth* $T(n)$ *is said to be* $U_{E'}$*-uniform if its extended connection language is recognizable by an alternating Turing machine with time bound* $O(\log(Z(n)))$.

Theorem 6 *Let a circuit family $\mathcal{F}$ have depth $T(n)$ and size $Z(n)$.*
If $T(n) = O(\log(Z(n)))$ and $\mathcal{F}$ is U_{E^*}-uniform, then $\mathcal{F}$ is $U_{E'}$-uniform.
Conversely, if $T(n) = \Omega(\log(Z(n)))$ and $\mathcal{F}$ is $U_{E'}$-uniform, then $\mathcal{F}$ is U_{E^*}-uniform.

Proof : Obvious.

Corollary 7 *If $Z(n) = n^{\Omega(1)}$ and $T(n)$ is both $\Omega(\log(n))$ and $O(\log(Z(n)))$, then the following are equivalent:*

- *Language $\mathcal{L}$ has U_E-uniform circuits of size $\Theta(Z(n))$ and depth $\Theta(T(n))$,*
- *Language $\mathcal{L}$ has U_{E^*}-uniform circuits of size $\Theta(Z(n))$ and depth $\Theta(T(n))$,*
- *Language $\mathcal{L}$ has $U_{E'}$-uniform circuits of size $\Theta(Z(n))$ and depth $\Theta(T(n))$.*

Proof : By Theorems 2 and 6.

We shall also sometimes assume, without loss of generality, that our circuits have fanout 1, i.e. each output bit has a disjoint tree of descendant gates. This may increase circuit size by a factor exponential in the depth, but none of the results of this paper depends on a smaller size bound than that, and in fact most of our circuits are that big anyway.

2.4 Function Growth Rates

An arbitrary function may have length depending on its inputs in very chaotic ways — for example, a function $f(i)$ might output the binary representation of the largest prime dividing the encoding of the i-th Turing machine that does not halt when given an empty input tape, preceded by a number of zeroes equal to the second-largest such prime. Practitioners in the field make a number of formal or informal assumptions to avoid such problems. Following is a relatively formal treatment of the assumptions common in the field. Our definitions are framed for "two-tier" functions, those that distinguish between normal and safe parameters, but they would work equally well in a more classical framework.

Definition 18 *The bit-graph of a function $f(\vec{x};\vec{y})$ is the set of tuples $\langle \vec{x},\vec{y},i,c\rangle$ such that $i < |f(\vec{x};\vec{y})|$ and $Bit(i, f(\vec{x};\vec{y})) = c$.*

Definition 19 *If $(x_1, x_2, \ldots x_k)$ and $(y_1, y_2, \ldots y_k)$ are tuples of numbers, we say $\vec{x} \leq \vec{y}$ if for all $1 \leq i \leq k$, $x_i \leq y_i$.*

Definition 20 *If $f(\vec{x};\vec{y})$ is a function with normal and safe inputs as specified, we define the* growth function of f *as the function $\mathcal{G}_f(\vec{x};\vec{y})$ equal to the maximum of $|f(\vec{x}';\vec{y}')|$, taken over all tuples of bit-strings $(\vec{x}',\vec{y}') \leq (\vec{x},\vec{y})$.*

Definition 21 *A function $f(\vec{x};\vec{y})$ is* smooth *if its length is a nondecreasing function of the lengths of $\vec{x},\vec{y}$, independent of the bits of $\vec{x},\vec{y}$.*

Definition 22 *If $f(\vec{x};\vec{y})$ is a function with length function L_f as above, we denote by $\overline{f}(\vec{x};\vec{y})$ the function whose value, for all $\vec{x},\vec{y} \in \{0,1\}^*$, is the value of $f(\vec{x};\vec{y})$ padded on the left with sufficient zeroes to make its length exactly $\mathcal{G}_f(\vec{x};\vec{y})$.*

Proposition 8 *For any function f,*

- *The growth function of f is identical to that of $\overline{f}$.*
- *$\overline{f}$ is smooth.*
- *If f is smooth then $\overline{f} \equiv f$.*
- *The bit-graph of f is FO-reducible to the Cartesian product of the bit-graphs of $\overline{f}$ and $|f|$.*
- *The bit-graph of $\overline{f}$ is FO-reducible to the Cartesian product of the bit-graphs of f and $\mathcal{G}_f$.*

Proof : Obvious.

The function $\mathcal{G}_f$, and hence $\overline{f}$, is of course reducible to f alone, but the reduction appears to require quantifying over exponentially many tuples of bit-strings, and hence it cannot obviously be done in FO.

Henceforth we assume all our functions are smooth, so we seldom need to worry about growth functions explicitly. One exception is the following definition:

Definition 23 *Let* $f(\vec{x};\vec{y})$ *be a function with normal and safe inputs as specified and with growth function* $\mathcal{G}_f$*. If* A *is an alternating Turing machine algorithm that computes* f*'s bit-graph, and* $\vec{a}$ *and* $\vec{b}$ *are suitable tuples of bit-strings, then* $d_A[\vec{a};\vec{b}]$ *is the maximum over all tuples* $(\vec{a}',\vec{b}') \leq (\vec{a},\vec{b})$ *and over all* $i < \mathcal{G}_f(\vec{a},\vec{b})$ *of the time necessary to compute bit* i *of* $f(\vec{a}';\vec{b}')$ *by alternating algorithm* A.

If F *is a family of one-output circuits that computes* f*'s bit-graph, then* $d_F[\vec{a};\vec{b}]$ *is the maximum over all* $i < \mathcal{G}_f(\vec{a};\vec{b})$ *of the depth of the circuit applicable to inputs* $\vec{a}$ *and* $\vec{b}$ *and bit position* i.

If F *is a family of multi-output circuits that computes* f*, then* $d_F[\vec{a};\vec{b}]$ *is the depth of the circuit applicable to inputs* $\vec{a}$ *and* $\vec{b}$.

Note, in all these cases, that $d[\vec{a};\vec{b}]$ *depends only on the* lengths *of* $\vec{a}$ *and* $\vec{b}$*, not on their* values.

Definition 24 *If* $f(\vec{x};\vec{y})$ *is computable by* $U_{E'}$*-uniform, bounded-fanin, polynomial-size, multi-output circuit family* F*, and there is a constant* c *such that for all* $\vec{a},\vec{b}$ *we have* $d_F[\vec{a};\vec{b}] \leq c \cdot \max(\log(|\vec{a}|,|\vec{b}|))$*, then we say* $f \in FNC^1$.

Similarly, if f *is computable by a* $U_{E'}$*-uniform, bounded-fanin, poly-size, multi-output circuit family* F *and there is a constant* c *such that for all* $\vec{a},\vec{b}$ *we have* $d_F[\vec{a};\vec{b}] \leq c$*, then we say* $f \in FNC^0$.

Chapter 3

Characterizing Log-depth Circuits

The main result of this chapter is the

Theorem 9 *For any smooth function $f(\vec{x};)$, the following are equivalent:*

(1) $f(\vec{x};) \in vsc(BASE)$

(2) $f(\vec{x};)$ *has $U_{E'}$-uniform, log-depth, polynomial-size circuits*

(3) the language $\{(i, b, \vec{x}) | Bit(i, f(\vec{x};)) = b\}$ *is recognized by an alternating $O(\log)$-time machine, and f has polynomial growth rate.*

Proof : We first prove that (1) implies f has polynomial growth rate (Lemma 10) and can be computed bitwise by log-depth, uniform circuits (Lemma 12); (2) follows from these two facts. The equivalence ($2 \Leftrightarrow 3$) follows from Theorems 3 and 4 (due to Ruzzo [Ruz81]). And we prove ($3 \Rightarrow 1$) by simulating an arbitrary ALOGTIME machine in $vsc(BASE)$ (Lemma 15), then simulating polynomial-length comprehension (Lemma 17) to build the function f from its bit-graph.

3.1 Bounding function growth

Lemma 10 *If $f(\vec{x}; \vec{y}) \in vsc(BASE)$, then there is a polynomial p_f with nonnegative coefficients such that*

$$|f(\vec{x}; \vec{y})| \leq p_f(|\vec{x}|) \cdot \max(|\vec{y}|, 1)$$

Note: The 1 is included to make things behave properly in case there are no "safe" parameters $\vec{y}$, or they are all zero-length.

Corollary 11 *If* $f(;\vec{y}) \in vsc(BASE)$*, then* f *has linear growth rate.*

This corollary is in fact the key to the proof, since the iterated function in very safe DCR must have no normal parameters: when a function of growth rate $n \mapsto kn$ is iterated to $\log(|z|)$ depth, the result has length $k^{\log(|z|)}n \doteq |z|^{\log(k)}n$, which is polynomial.

Proof : The fastest-growing functions in BASE are $Conc(;x,y)$, $Ins_0(;x,y)$, and $Ins_1(;x,y)$.

$$\begin{aligned} |Conc(;x,y)| &= |x| + |y| \\ &\leq 2 \cdot \max(|x|,|y|) \end{aligned}$$

so the $Conc$ function satisfies the statement of the Lemma with $p_f \equiv 2$; Ins_0 and Ins_1 are similar. All the remaining BASE functions have output no longer than their longest (safe) input, so $p_f \equiv 1$ will suffice for them.

Definition by Safe Composition: Let $f(\vec{x};\vec{y}) = g(u(\vec{x};);v(\vec{x};\vec{y}))$, and suppose p_g, p_u, and p_v are polynomials satisfying Lemma 10 for functions g, u, and v respectively. Then

$$\begin{aligned} |f(\vec{x};\vec{y})| &\leq p_g(|u(\vec{x};)|) \cdot |v(\vec{x};\vec{y})| \\ &\leq p_g(p_u(|\vec{x}|)) \cdot p_v(|\vec{x}|) \cdot \max(|\vec{y}|,1) \end{aligned}$$

so it would suffice to let $p_f(|\vec{x}|) = p_g(p_u(|\vec{x}|)) \cdot p_v(|\vec{x}|)$. The proof for multiple functions $\vec{u}$ and $\vec{v}$ is similar.

Definition by Very Safe DCR: Let

$$f(z,b,\vec{x};\vec{y}) = \\ = \begin{cases} g(z,\vec{x};\vec{y}) & \text{if } |z| \leq \max(|b|,1) \\ h(;z,\vec{x},\vec{y},f(Fh(;z),b,\vec{x};\vec{y}),f(Bh(;z),b,\vec{x};\vec{y})) & \text{if } |z| > \max(|b|,1) \end{cases}$$

and suppose we know, by an inductive assumption, that (for some polynomial p_g and constant k_h)

$$|g(z, \vec{x}; \vec{y})| \leq p_g(|z|, |\vec{x}|) \cdot \max(|\vec{y}|, 1)$$

and

$$|h(; z, \vec{x}, \vec{y}, u_1, u_2)| \leq k_h \cdot \max(|z|, |\vec{x}|, |\vec{y}|, |u_1|, |u_2|)$$

Then we prove, by induction on $|z|$, that

$$|f(z, b, \vec{x}; \vec{y})| \quad \leq \quad k_h^{\lceil \log(|z|) \rceil} \cdot \max(|z|, |b|, |\vec{x}|, p_g(|b|, |\vec{x}|) \cdot \max(|\vec{y}|))$$

Note that $k_h^{\lceil \log(|z|) \rceil} \leq k_h^{\log(|z|)+1} = k_h \cdot |z|^{\log(k_h)}$ is polynomial in $|z|$, so the entire bound is polynomial in $|z|, |b|, |\vec{x}|$ and linear in $\max(|\vec{y}|)$ as desired.

The base case, for $|z| \leq |b|$, is obvious. For larger $|z|$, we know

$$\begin{aligned}
|f(z, b, \vec{x}; \vec{y})| &\leq k_h \cdot \max(|z|, |\vec{x}|, |\vec{y}|, |f(Bh(; z), b, \vec{x}; \vec{y})|, |f(Fh(; z), b, \vec{x}; \vec{y})|) \\
&\leq k_h \cdot \max(|z|, |\vec{x}|, |\vec{y}|, \\
&\quad k_h^{\lceil \log(\lceil |z|/2 \rceil) \rceil} \cdot \max(\lceil |z|/2 \rceil, |b|, |\vec{x}|, p_g(|b|, |\vec{x}|) \cdot \max(|\vec{y}|))) \\
&\leq k_h^{1+\lceil \log(\lceil |z|/2 \rceil) \rceil} \cdot \max(|z|, |b|, |\vec{x}|, p_g(|b|, |\vec{x}|) \cdot \max(|\vec{y}|)) \\
&= k_h^{1+\lceil \log(|z|) \rceil - 1} \cdot \max(|z|, |b|, |\vec{x}|, p_g(|b|, |\vec{x}|) \cdot \max(|\vec{y}|)) \\
&\quad \text{(If } |z| \text{ is even, then } \log\left(\left\lceil \tfrac{|z|}{2} \right\rceil\right) = \log(|z|/2) = \log(|z|) - 1, \\
&\quad \text{while if } |z| \text{ is odd, then } \lceil \log(|z|) \rceil = \lceil \log(|z| + 1) \rceil.\text{)} \\
&= k_h^{\lceil \log(|z|) \rceil} \cdot \max(|z|, |b|, |\vec{x}|, p_g(|b|, |\vec{x}|) \cdot \max(|\vec{y}|))
\end{aligned}$$

as desired. ∎

3.2 Bounding circuit depth

We next state a similar lemma about the circuit depth necessary to compute functions in $vsc(BASE)$:

Lemma 12 *If $f(\vec{x};\vec{y}) \in vsc(BASE)$, then there are constants k_f and c_f and a $U_{E'}$-uniform circuit family F computing the bit-graph of f with depth bound*

$$d_F[\vec{x};\vec{y}] \leq k_f \cdot \max(||\vec{x}||) + c_f.$$

Corollary 13 *If $f(;\vec{y}) \in vsc(BASE)$, then f can be computed by a $U_{E'}$-uniform family of constant-depth circuits; that is, it's in FNC^0.*

Again the corollary is in fact the key to the proof: when a function of constant circuit depth is iterated $\log(|z|)$ times, the result clearly has depth $O(\log(|z|))$.

In this section, we in fact only prove the depth bounds; then, in Section 3.3, we show how to make the circuits thus constructed $U_{E'}$-uniform.

Proof: The BASE functions all have extremely simple circuits. The circuits for $\lambda, 0$, and 1 are in fact constant-valued, while the functions $\pi_k^{i,j}$, MSP, LSP, $Conc$, Fh, Bh, Ins_0, and Ins_1 can each be computed by wires routing inputs to outputs, with no gates at all. So all these functions have constant-depth, bounded-fanin circuits. $Cond$, Or, and Not actually use some gates, but they too are clearly in FNC^0.

Definition by Safe Composition: Suppose $f(\vec{x};\vec{y}) = g(u(\vec{x};);v(\vec{x};\vec{y}))$ and that functions g, u, and v are computed by uniform circuit families G, U, and V respectively. Construct the circuit family F in the obvious way: an instance of circuit family G, with the input gates tied to the outputs of instances of circuit families U and V. Now suppose that k_g, c_g, k_u, c_u, k_v, and c_v are constants satisfying Lemma 12 for circuit families G, U, and V. Finally, suppose (by Lemma 10) p_u is a polynomial such that $|u(\vec{x};)| \leq p_u(|\vec{x}|)$. Since $p_u(|\vec{x}|)$ is bounded by a monic polynomial in $\max(|\vec{x}|)$, we assume without loss of generality that it actually is one, i.e. $p_u(|\vec{x}|) = a_u \cdot \max(|\vec{x}|)^{\deg(p_u)}$. Then

$$\begin{aligned}
&d_F[\vec{x};\vec{y}] \leq \\
&\quad\leq\ d_G[u(\vec{x};);v(\vec{x};\vec{y})] + \max(d_U[\vec{x};], d_V[\vec{x};\vec{y}]) \\
&\quad\leq\ k_g \cdot ||u(\vec{x};)|| + c_g + \max(k_u \cdot \max(||\vec{x}||) + c_u, k_v \cdot \max(||\vec{x}||) + c_v) \\
&\quad\leq\ k_g \cdot |a_u \cdot \max(|\vec{x}|)^{\deg(p_u)}| + c_g + \max(k_u \cdot \max(||\vec{x}||) + c_u, k_v \cdot \max(||\vec{x}||) + c_v)
\end{aligned}$$

$$\leq \quad k_g \cdot \deg(p_u) \cdot \max(||\vec{x}||) + |a_u| + c_g + \max(k_u, k_v) \cdot \max(||\vec{x}||) + \max(c_u, c_v)$$
$$\leq \quad (k_g \cdot \deg(p_u) + \max(k_u, k_v)) \cdot \max(||\vec{x}||) + \max(c_u, c_v) + |a_u| + c_g$$

which is, as desired, linear in $\max(||\vec{x}||)$. The proof in the case that there are multiple functions $\vec{u}$ and $\vec{v}$ is similar.

Definition by Very Safe DCR: Suppose

$$f(z, b, \vec{x}; \vec{y}) =$$
$$= \begin{cases} g(z, \vec{x}; \vec{y}) & \text{if } |z| \leq \max(|b|, 1) \\ h(; z, \vec{x}, \vec{y}, f(Fh(; z), b, \vec{x}; \vec{y}), f(Bh(; z), b, \vec{x}; \vec{y})) & \text{if } |z| > \max(|b|, 1) \end{cases}$$

and that k_g, c_g, and c_h satisfy the present lemma for circuit families G and H computing g and h respectively. Let F denote the obvious circuit family for f: an instance of circuit H, replacing each of its inputs by either a smaller instance of F or an instance of G, depending on whether $|z| > |b|$. Then for $|z| > |b|$,

$$d_f[z, b, \vec{x}; \vec{y}] \leq$$
$$\leq \quad d_H[; z, \vec{x}, \vec{y}, f(Fh(; z), b, \vec{x}; \vec{y}), f(Bh(; z), b, \vec{x}; \vec{y})] + d_F[Bh(; z), b, \vec{x}; \vec{y}]$$
$$\leq \quad c_h + d_F[Bh(; z), b, \vec{x}; \vec{y}]$$
$$\leq \quad ||z|| \cdot c_h + d_G[z, \vec{x}; \vec{y}]$$
$$\leq \quad ||z|| \cdot c_h + k_g \cdot \max(||z||, ||\vec{x}||) + c_g$$
$$\leq \quad (c_h + k_g) \cdot \max(||z||, ||\vec{x}||) + c_g$$

∎

Thus each bit of $f(\vec{x}; \vec{y})$ can be computed in log depth (and therefore polynomial size). Since there are only polynomially many bits of output, the whole value of f can be computed by NC^1 circuits. It remains to show that these circuits are sufficiently uniform.

3.3 Input-output mappings

In order to conclude that every $\{0,1\}$-valued function in $vsc(BASE)$ is in ALOGTIME, we need our circuits to be not only log-depth but $U_{E'}$-uniform, i.e. to have an extended connection language in ALOGTIME. In order to get the desired uniformity, we impose a stronger induction hypothesis on our circuit families. Describing the additional requirements calls for some preliminary definitions.

We construct a term algebra to operate on bit positions, capable of representing such information as "bit 7 output from this circuit depends on bit 4 of parameter y_1 and bit 6 of parameter y_3." Of course, in some circuits there are output bits that depend on no inputs whatsoever — for example, the unconditionally-0 even-numbered bits of output from the Ins_0 function — and to represent this fact we use a special symbol $\perp$ to indicate "undefined bit position". This "undefined" object $\perp$ has the property that any operator applied to it yields $\perp$.

Definition 25 *An* extended index *is either an integer i or the special symbol $\perp$. An* extended index in string x *is either $\perp$ or a natural number $0 \leq i < |x|$.*

Definition 26 *The* mapping language *for a fixed set of parameters $y_1, y_2, \ldots y_k$, denoted $\mathcal{ML}(\vec{y})$, is defined to be the set of terms constructed from the constant symbol i and the following unary operators over the universe of extended indices:*

$$\begin{aligned}
UNDEF(x) &= \perp, \text{ regardless of } x \\
ZERO(x) &= 0, \text{ regardless of } x \text{ unless } x = \perp \\
ADD_j(x) &= x + |y_j|, \text{ for } 1 \leq j \leq k \\
SUB_j(x) &= x - |y_j|, \text{ for } 1 \leq j \leq k \\
ADDHALF_j(x) &= x + \left\lceil \frac{|y_j|}{2} \right\rceil, \text{ for } 1 \leq j \leq k \\
HALVE(x) &= \left\lfloor \frac{x}{2} \right\rfloor
\end{aligned}$$

Since there is only one constant symbol in this language, and all operators are unary, every closed term can be thought of as a list of symbols from the $3k+3$-symbol alphabet $\{UNDEF, ZERO, ADD_j, SUB_j, ADDHALF_j, HALVE\}$, followed by an i, while an open term is simply such a list *not* followed by an i. Composition of open terms can then be done simply by concatenating the corresponding lists. The empty term is naturally interpreted as the identity map.

Lemma 14 *For any bit-strings $y_1, y_2, \ldots y_k$ of lengths $\leq n$, there is an alternating Turing machine which, given an open term $t \in \mathcal{ML}(\vec{y})$ containing m operators, and a value $i \leq n$, computes the bit-graph of the term $t(i)$ in time $O(\log(m) + \log\log(n))$.*

Proof : We prove this first for a more restricted language, then show step by step how to deal with the operators of the full language. If the operators $HALVE$, $ZERO$, and $UNDEF$ do not appear, each operator in the term corresponds to adding a constant to, or subtracting a constant from, a running total which begins with i. Using two's-complement notation for negatives, we can treat this as simply adding up a list of m binary numbers, each of length $\log(n)$. Each ADD_j operator corresponds to a $|y_j|$ in the list, each SUB_j to a $-|y_j|$, each $ADDHALF_j$ to a $\left\lceil \frac{|y_j|}{2} \right\rceil$, and the i at the end to the value of i; any desired bit of any desired list entry can be computed in time $O(\log(m) + \log\log(n))$, since $\vec{y}$ are treated as constants. We can then add the elements of the list by the well-known carry-save algorithm for vector addition, taking alternating time $O(\log(m) + \log\log(n))$.

If the $HALVE$ operator appears in the language, but $ZERO$ and $UNDEF$ are still disallowed, we can evaluate the term as follows: for each operator in parallel, count the number of $HALVE$ operators appearing after it (this can be done in alternating $O(\log(m))$ time, for example by carry-save addition on a vector of 0's and 1's). Then construct a list of binary numbers as before from the non-$HALVE$ operators, but left-shift each entry in the list by the number of $HALVE$ operators appearing after it. (Since we're only trying to define a bit-graph, this left-shift can be done by subtracting bit positions, in time $O(\log(\max(m, \log(n))))$.) Each entry in the list is now possibly as long as $O(m + \log(n))$ bits, but the carry-save addition algorithm can still add up the list in alternating time $O(\log(m) + \log\log(n))$. Now right-shift the result by the number of $HALVE$ operators in the whole list, discarding the low bits shifted out. This too requires only addition of bit positions, and so takes time $O(\log(m + \log(m) + \log(n))) = O(\log(m) + \log\log(n))$.

Adding the $ZERO$ operator to the language, we can still evaluate the term as follows: guess (in alternating time $O(\log(m))$) the location of the first $ZERO$ operator, confirm by universal branching (another $O(\log(m))$ alternating time) that there are none before it, discard all of the term following the $ZERO$ operator, set $i = 0$, and use the above algorithm.

Finally, the $UNDEF$ operator is easily handled: if there is an $UNDEF$ operator in the term (which can be checked by an existential guess in alternating time $O(\log(m))$), output $\perp$; otherwise use the above algorithm. ∎

For example, the term

$$ADD_3(HALVE(ADDHALF_1(SUB_3(HALVE(ADD_2(ZERO(SUB_1(i))))))))$$

would be evaluated as follows: find the $ZERO$ operator and replace it and everything following it by 0. Then construct the list

$$\left(4|y_3|, 2\left\lceil\frac{|y_1|}{2}\right\rceil, -2|y_3|, |y_2|, 0\right),$$

add its elements, and right-shift the result by two bits.

We shall use the above lemma with m proportional to the depth of an NC^1 circuit, so $m = O(\log(|\vec{y}|))$, and the lemma tells us we can evaluate a term of this length in alternating $O(\log\log(|\vec{y}|))$ time; in fact, $O(\log(|\vec{y}|))$ time would suffice for our purposes.

To make this language, and the evaluation of terms in it, more concrete, we observe that for each of the BASE functions, the constantly-many bits of input that affect any given output bit are described by terms in the language in combination with finitely much information to select a parameter.

- λ has no output bits.
- 0 and 1 are unaffected by input, so term $UNDEF(i)$ indicates which bit of input affects output bit i (i.e. none).
- Bit i of $MSP(;x,y)$ is bit $i+|y|$ of x, so the pertinent input is bit $ADD_2(i)$ of the first parameter.
- Bit i of $LSP(;x,y)$ is bit i of x, if this exists, and otherwise the constant 0, which depends on no input at all.
- Bit i of $Cond(;x,y,z)$ depends on bit 0 of x, bit i of y, and bit i of z.
- Bit i of $Conc(;x,y)$ is either bit i of y or bit $i-|y|$ of x, whichever one exists; that is, bit $SUB_2(i)$ of parameter 1 and bit i of parameter 2 affect the output.
- Bit i of $Bh(;x)$ is bit i of x, so long as i is in the output at all.
- Bit i of $Fh(;x)$ is bit $i+\left\lceil\frac{|x|}{2}\right\rceil$ of x, so the term is $ADDHALF_1(i)$.
- Bit i of $Not(;x)$ depends on bit i of x.
- Bit i of $Or(;x,y)$ depends on bit i of x and bit i of y.
- Bit i of $Ins_0(;x)$ and of $Ins_1(;x)$ depends on bit $\left\lfloor\frac{i}{2}\right\rfloor = HALVE(i)$ of x, if i is odd, and on no input whatsoever, i.e. $UNDEF(i)$, if i is even.

This fact motivates the following definition.

Definition 27 *A family of depth d circuits with input parameters $y_1, y_2, \ldots y_k$ is* mapping-uniform *if there is a constant-time algorithm which, given a d-bit string σ of L's and R's, produces a term t and a number $1 \leq j \leq k$ such that for all i, the descendant of output bit i along the path σ is an input gate tied to bit $t(i)$ of parameter y_j.*

All the BASE functions have circuits that are both $U_{E'}$- and mapping-uniform. For BASE functions λ, 0, 1, MSP, Bh, Fh, and Not this is easy: there's only one term that describes the dependency, so the "constant-time algorithm" is to write down that term regardless of σ. The natural circuit for Or is a layer of OR gates with left child from x and right child from y, so the following constant-time algorithm works: output the empty term unconditionally, and given a length-1 path σ, output $j = 1$ (indicating bit i of x) if $\sigma = L$, otherwise $j = 2$ (indicating bit i of y).

For the remaining functions it's a little trickier. The natural circuit for $Cond$ consists of a layer each of two-input ORs and two-input ANDs. Which of the four possible paths one follows through this circuit easily determines whether the input should be bit i of y, bit i of z, bit 0 of x, or the negation of bit 0 of x.
Ins_0 can be made mapping-uniform by making the gate tied to output i either an AND (if i is even) or an OR (if i is odd), whose two inputs in either case are a constant 0 and bit $\left\lfloor \frac{i}{2} \right\rfloor$ of the parameter. We have effectively dumped the problem of recognizing whether i is even or odd onto the ALOGTIME algorithm (from $U_{E'}$-uniformity) that checks gate types (since this constant-time algorithm has no access to i). Ins_1 is similar, with 0 replaced by 1 and the rôles of OR and AND reversed.
We handle LSP similarly: let the left child of each output gate be a 0 and the right child a corresponding bit of input, and make the gate either an AND (and output 0) or an OR (and output the input bit), depending on the bit position. Note that in this case the term for right children may specify bits of input that don't actually exist, but this only occurs in situations in which the value of the bit doesn't matter. The same happens for $Conc$: the output gates are the outputs of two-input gadgets computing projection functions, and the gate-type algorithm is responsible for deciding which projection function to use, the other input being meaningless.

Now let us consider the uniformity of these circuits constructed in the proof of Lemma 12. We prove by induction on function definitions that all the circuits are $U_{E'}$-uniform, and furthermore that circuits for functions with no normal parameters can be made mapping-uniform as well. As shown above, all the BASE functions have constant-depth, $U_{E'}$- and mapping-uniform circuits. We must show that safe composition preserves $U_{E'}$ uniformity (and, if applicable, mapping-uniformity), and that very safe DCR, assuming the iterated function has $U_{E'}$- and mapping-uniform circuits, produces a $U_{E'}$-uniform result.

Suppose a function $f(\vec{x};\vec{y})$ is defined by safe composition as $g(u(\vec{x};);v(\vec{x};\vec{y}))$, and that we have circuit families G, U, and V satisfying the inductive hypotheses and computing functions g, u, and v respectively. We construct an instance of G, with its inputs tied to the outputs of instances of U and V, with the latter replicated as necessary to make the whole circuit treelike. Let s be the size of the largest U or V subcircuit appearing in this circuit, and let r be such that $2^{r-1} < s \leq 2^r$. Now renumber each gate of the U and V subcircuits by concatenating the number of the G-input gate from which it descends with its own r-bit gate number and an extra 1 bit; we similarly renumber each gate of the G subcircuit by concatenating $r+1$ zeroes to the end of its number.

We must now show the extended connection language of the resulting circuit family F in ALOGTIME. Given the number of a gate in F, we can determine its type as follows. If the low bit is 0, delete the low $r+1$ bits and invoke the uniformity algorithm for G. If the low bit is 1, ignoring the low $r+1$ bits again yields the gate number of an input gate to G; the uniformity algorithm for G tells us which input parameter it is and hence whether the following r bits indicate a U or a V circuit; we can then invoke the U or V uniformity algorithm on those r bits to decide the gate type. If this type indicates a Boolean function, we're done; otherwise it indicates an input to U or V.

On the other hand, given the numbers of two gates a and b in F and a log-length path of L's and R's, we can check whether gate a is the descendant of gate b along the specified path as follows. If both a and b are in the G portion of the circuit, delete their last $r+1$ bits and use the G uniformity algorithm. If both are in a U or V portion of the circuit, check that both descend from the same G-input gate, and then use the U or V uniformity algorithm. If a is in a U or V subcircuit and b is in the G subcircuit, let c be the unique input gate to G which is descended from b along a prefix of the path; this gate c can be found by the G uniformity algorithm. By the G uniformity algorithm again, find out which input bit it is tied to, then use the U or V uniformity algorithm to see whether gate a is descended along the remainder of the path from the corresponding output gate of the U or V circuit. The proof for multiple functions u and v is similar.

We must also show that composition of mapping-uniform circuits preserves mapping-uniformity. In this case, for concreteness, we shall explicitly consider what happens with multiple functions: suppose $f(;y_1, y_2, \ldots y_k) = g(;u(;\vec{y}), v(;\vec{y}), \vec{y})$ and G, U, V are mapping-uniform circuit families computing g, u, and v respectively. We want to show that the natural circuits for F are also mapping-uniform. We have already proven that F, G, U, and V are constant-depth, so mapping-uniformity makes sense. Let d_G, d_U, and d_V be the depths of circuits in G and V, and let $d_F = d_G + \max(d_U, d_V)$; assume without loss of generality that all paths in the circuits are padded to be exactly those lengths. Now given a d_F-bit $\{L, R\}$ path σ, call the first d_G bits σ_G. By the mapping-uniformity of G, we can compute a term t_G and a number $1 \leq j_G \leq k+2$ such that for all i, the G-input gate reached from output position i of G by path σ_G is input bit $t_G(i)$ of parameter j_G of g. In the present example parameter 1 of g is $u(;\vec{y})$, parameter 2 is $v(;\vec{y})$, and parameters 3 through $k+2$ are inputs y_1 through y_k respectively. If $j_G = 1$, then, let the remaining bits of σ be called σ_U and use the mapping-uniformity of U to compute a term t_U and a parameter number $1 \leq j_U \leq k$ such that the gate descended from output bit i by path σ is input $t_U(t_G(i))$ of y_{j_U}. If $j_G = 2$, do likewise with V in place of U. If $j_G > 2$, the input is tied to bit $t_G(i)$ of y_{j_G-2}. Observe that everything we've done here is constant-time: compute j_G and branch by cases on it, producing as output one of the ordered pairs $(t_U \circ t_G, j_U)$, $(t_V \circ t_G, j_V)$, or $(t_G, j_G - 2)$. The proof for more or fewer functions u, v is similar.

Now suppose $f(z, b, \vec{x}; \vec{y})$ is defined by very safe DCR:

$$f(z, b, \vec{x}; \vec{y}) =$$
$$= \begin{cases} g(z, \vec{x}; \vec{y}) & \text{if } |z| \leq \max(|b|, 1) \\ h(; z, \vec{x}, \vec{y}, f(Fh(; z), b, \vec{x}; \vec{y}), f(Bh(; z), b, \vec{x}; \vec{y})) & \text{if } |z| > \max(|b|, 1) \end{cases}$$

Since h takes no normal parameters, it has constant-depth, $U_{E'}$- and mapping-uniform circuits H, and by assumption G has log-depth, $U_{E'}$-uniform circuits G. Each instance of the natural circuit family F comprises a $\log(|z|)$-depth binary tree of H subcircuits and a layer of G subcircuits at the bottom. Each instance of H or G is uniquely determined, within the circuit, by the $\log(|z|)$-length sequence of Bh and Fh functions determining what portion of z the H or G circuit is given. As usual, we assume the circuits are treelike, and let $n = \max(|z|, |b|, |\vec{x}|, |\vec{y}|)$. We number each gate in such a circuit by concatenating three $O(\log(n))$-bit numbers (suitably delimited to make parsing easy): the output bit (of the whole circuit) from which it descends, a string of 0's and 1's indicating the sequence of Bh and Fh functions identifying the G or H subcircuit in which the gate resides, and the "local" gate number of the gate within its own G or H subcircuit.

We need to show that the extended connection language of F, with gates numbered as above, is in ALOGTIME. Given a gate number, we can easily determine its gate type by simply examining the portion of the gate number that corresponds to a local gate number and applying the G or H uniformity algorithm. If the gate is internal to its local subcircuit, it computes some Boolean function and we're done. If the gate is an input to its local subcircuit but not an input to the whole F circuit, it must be tied to the output of a lower-level H or G circuit; in this case we can simply say the gate computes the identity function (or we can make it an AND with two identical inputs, or something of the sort). Finally, if the gate is tied to, say, bit i of non-recursive parameter j, we are again done unless $j = 1$, in which case we're looking at bit i of the substring of z determined by the $\{Bh, Fh\}$ path. We compute what bit of z this corresponds to as follows: compute $|z|$, then follow the $\{Bh, Fh\}$ path, halving $|z|$ each time and rounding up or down as appropriate. In addition, whenever an Fh is reached, not only replace $|z|$ by $\left\lfloor \frac{|z|}{2} \right\rfloor$ but write the value $\left\lceil \frac{|z|}{2} \right\rceil$ on a list. When the path is exhausted, the list contains $O(\log(|z|))$ numbers of $O(\log(|z|))$ length whose sum is the offset of the relevant substring of z from z itself. Compute the sum by the carry-save addition algorithm in ALOGTIME and add the result to i to determine to which bit of z the gate is tied.

On the other hand, given two gate numbers a and b and a log-length path σ of $\{L, R\}$ decisions, we can determine whether gate a is descended from gate b by the specified path as follows. First, if a and b are not descended from the same output gate of F, the answer is "no". If the $\{Fh, Bh\}$ paths in their gate numbers are identical, they both belong to the same G or H subcircuit, so we can simply apply the uniformity algorithm for G or H. If the $\{Fh, Bh\}$ path of b is not a prefix of that of a, the answer is "no". Finally, if the $\{Fh, Bh\}$ path in b's gate number is a prefix of that in a's gate number, we guess gates a' and b' such that b' is an input to the H circuit in which b resides, a' is an output from the G or H circuit in which a resides, b' is descended from b by a prefix of the path, and a is descended from a' by a suffix of the path; all this can be checked in ALOGTIME by the G and H uniformity algorithms. Follow the intervening part of the $\{L, R\}$ path, applying the mapping-uniformity of H to construct a sequence of terms t_r and parameter numbers j_r, taking deterministic time $O(\log(|z|))$ in total because each step takes constant time. (In fact, each could be computed in parallel, but this buys us nothing because we'll eventually need term t, the composition of the t_r's, anyway.) If any j_r indicates the parameter z, b, or one of $\vec{x}$,$\vec{y}$, the answer is "no". Likewise, if some j_r indicates that the input to level r comes from the recursive call $f(Fh(;z), b, \vec{x}; \vec{y})$ but the corresponding bit of the $\{Fh, Bh\}$ path to a' indicates recursion on Bh, or *vice versa*, the answer is "no". Gate b' is input to an H subcircuit, and by the H uniformity algorithm we can determine (in ALOGTIME) which input bit i it is. We then evaluate the term $t(i)$ to compute which bit position gate a' should be. (Strictly speaking, Lemma 14 does not apply here, since it expects all the lengths of parameters to be constant, while in the present setting any ADD_1, SUB_1, or $ADDHALF_1$ operator in fact involves a substring of

z that varies with level. But the ALOGTIME algorithm in the previous paragraph for computing the offset between a substring of z and the whole string z can easily be modified to an ALOGTIME algorithm for computing the *length* of a substring of z, and this algorithm is easily incorporated into the algorithm of Lemma 14.) Using the G or H uniformity algorithm we check that a' is in fact that bit of output, and we're done.

Thus the extended connection language of F is in ALOGTIME. This proves one direction of Theorem 9.

3.4 Simulating Turing machines

Lemma 15 *Given an alternating Turing machine M with inputs $\vec{X}$ and time bound $t = O(\log(|\vec{X}|))$, there is a function $f(\vec{X};) \in vsc(BASE)$ such that*

$$f(\vec{X};) = \begin{cases} 1 & \textit{if } M \textit{ accepts } \vec{X} \\ 0 & \textit{if } M \textit{ rejects } \vec{X} \end{cases}$$

In other words, $vsc(BASE)$ can simulate the computation of an arbitrary ALOGTIME machine.

The proof of this lemma is algorithmic and fairly technical, and takes up most of the rest of Chapter 3.

Proof of Lemma 15: We start by defining a number of utility functions to manipulate strings. The first is

$$ZEROES(;x) = LSP(;\lambda, x)$$

which returns a string of zeroes of the same length as x.

Next we define

$$PADRT(;x, y) = Conc(;x, ZEROES(;MSP(;y, x)))$$

which pads its argument x on the right with zeroes to length $\max(|x|, |y|)$.

We frequently wish to get at the leftmost or rightmost bit of a string.

$$\begin{aligned}
LOBIT(;x) &= LSP(;x,1) \\
DELLOBIT(;x) &= MSP(;x,1) \\
HIBIT(;x) &= MSP(;x,MSP(;x,1)) \\
DELHIBIT(;x) &= LSP(;x,MSP(;x,1))
\end{aligned}$$

We also define a function to shift a number without changing its length.

$$SHL(;x,y) \quad = \quad LSP(;Conc(;x,y),x)$$

shifts x left by appending y, discarding the high bits so the result is the same length as x.

We would also like to be able to find the length of a string.

$$Len(x;) = \begin{cases} \lambda & \text{if } |x| = 0 \\ 1 & \text{if } |x| = 1 \\ Conc(;Len(Fh(;x);),1) & \text{if } |x| > 1 \end{cases}$$

returns a string y such that $|y| = ||x||$. Note that $Len(x;)$ does *not* return a binary representation of $|x|$ itself.

We would frequently like to be able to assume, without loss of generality, that the length of some string is exactly a power of 2. This can be justified (with thanks to Andrew Klapper for insisting that I justify it!) as follows: define

$$\begin{aligned}
&ROUNDUP(x;) = \\
&= \begin{cases} 1 & \text{if } |x| \leq 1 \\ Conc(;ROUNDUP(Bh(;x);),ROUNDUP(Bh(;x);)) & \text{otherwise} \end{cases}
\end{aligned}$$

so $ROUNDUP(x;)$ is a string of 1's whose length is the least power of 2 greater than or equal to $|x|$. Then we can define

$$EXTEND(x;) = LSP(;x,ROUNDUP(x;))$$

which produces x padded on the left with zeroes up to power-of-2 length.

Since a log-time Turing machine is defined with random access to its read-only input tape, we need a function to implement random access. We define the function $GETBIT(X;i)$ to return bit i of the normal parameter X, treating i as a binary number. By convention, if $|i| > \log(|X|)$ we shall only pay attention to the low $\log(|X|)$ bits of i, while if $|i| < \log(|X|)$ we pad i on the left with zeroes. The function is defined by simultaneous, very safe DCR: let

$$GETBIT(X;i) = f_1(X; EXTEND(X;), LSP(;i, DELLOBIT(;Len(X;))))$$

where

$$f_1(Y;X,i) = \\ = \begin{cases} X & \text{if } Y \leq 1 \\ Fh(;f_1(Fh(;Y);X,i)) & \text{if } Y > 1 \text{ and } HIBIT(;f_2(Fh(;Y);X,i)) = 1 \\ Bh(;f_1(Fh(;Y);X,i)) & \text{if } Y > 1 \text{ and } HIBIT(;f_2(Fh(;Y);X,i)) = 0 \end{cases}$$
$$f_2(Y;X,i) = \\ = \begin{cases} PADRT(;i,X) & \text{if } |Y| \leq 1 \\ SHL(;Fh(;f_2(Fh(;Y);X,i))),1) & \text{if } |Y| > 1 \end{cases}$$

Note that $|X| = |PADRT(;i,X)|$, so the equal-length requirement is met at the base level. It is also easily proven by induction that $|f_1(Y;X,i)| = \frac{|X|}{|Y|}$ is a power of 2 for all X, Y that arise in the recursion. In particular, $|f_1(Fh(;Y);X,i)|$ is always even, so if $|f_1(Fh(;Y);X,i)| = |f_2(Fh(;Y);X,i)|$ then the lengths of $Fh(;f_1(Fh(;Y);X,i))$, $Bh(;f_1(Fh(;Y);X,i))$, and $SHL(;Fh(;f_2(Fh(;Y);X,i)),1)$ are all equal. Thus this definition meets all the equal-length criteria of Lemma 5, and so $GETBIT \in vsc(BASE)$.

As for its correctness, the $GETBIT$ function first coerces i to have length $\log(|X|) = ||X|| - 1$, discarding high bits if i is too long and left-padding with zeroes if it's too short, and then goes into the simultaneous recursion. We consider f_1's value to be a portion of X containing the desired bit, and f_2's to be the suffix of i determining where in f_1 the desired bit is. At each step on the way up, the high bit of i determines whether to keep the front or back half of X; the other half is discarded, as is the high bit of i. By the time the functions return, i has been reduced to nothing and X to a single bit, which is the answer.

Next we'd like to build a binary tree of all the possible bit-strings of a given length t, to be used as paths from root to leaves. But for technical reasons we actually produce a tree not of paths but of "inflated paths", in which the bits of interest appear in bit positions $0, 1, 3, 7, \ldots 2^t - 1$, and the bits in between are ignored. (Each of 2^t paths is inflated to length 2^t, so the tree has length 2^{2t}, so we've only squared the size of the tree.) Producing this binary tree, which must contain a different inflated path at each leaf, with bitwise-similar leaves near one another in the tree and with an easy mechanism for distinguishing universal branches from existential ones, is the most complex part of the proof.

Definition 28 *If x is a bit-string, we define the* deflation *of x as the bit string y, of length $||x||$, such that for all $i < |y|$, $Bit(i, y) = Bit(2^i - 1, x)$. Conversely, we say any such x is an* inflation *of y. A string x is a* full inflation *of y if x is an inflation of y and $|x| = 2^{|y|-1}$.*

The function that produces this binary tree of inflated paths is defined as follows:

$$PATHS(x;) = \begin{cases} \lambda & \text{if } |x| = 0 \\ 01 & \text{if } |x| = 1 \\ h(; PATHS(Bh(; x);)) & \text{if } |x| > 1 \end{cases}$$

where

$$h(; v) = \begin{cases} Conc(; Ins_0(; v), Ins_1(; v)) & \text{if } HIBIT(Fh(v)) \neq HIBIT(Bh(v)) \\ Conc(; Ins_0(; v), Ins_1(; Not(; v))) & \text{if } HIBIT(Fh(v)) = HIBIT(Bh(v)) \end{cases}$$

Clearly each iteration of h quadruples the length of the function, so if $|x|$ is a power of 2,

$$|PATHS(x;)| = 2 \cdot 4^{\log(|x|)} = 2|x|^2$$

The value of $PATHS(x;)$ can be usefully interpreted as $2|x|$ blocks of length $|x|$. Indeed, it can be thought of as (the root of) a binary tree: define $N_k^{i,x}$, the k-th node of height i in $PATHS(x;)$, to be bits $k \cdot 2^i|x|$ through $(k+1)\cdot 2^i|x|-1$, as long as these numbers are less than $2|x|^2$, the total number of bits. Then the nodes of height 0, the leaves, are the individual blocks of length $|x|$ and the left and right children of any non-leaf node are its front and back halves. Furthermore, the following are all equivalent:

- $N_j^{i,x}$ is a descendant of $N_{j'}^{i',x}$
- $j'2^{i'}|x| \le j2^i|x| < (j+1)2^i|x| \le (j'+1)2^{i'}|x|$
- $j'2^{i'} \le j2^i < (j+1)2^i \le (j'+1)2^{i'}$

Note that this last inequality is independent of x.

The less obvious properties of $PATHS$ are presented in the following technical lemma. The notation $\pm x$ denotes "either x or $Not(;x)$".

Lemma 16 *Let $|x|$ be a power of 2 and $i \le \log(2|x|) = ||x||$. Then*

(1) If $|x| > 1$, and if $j \ge \lfloor |x|/2^i \rfloor$, then $N_j^{i,x} = Ins_0(;N^{i,Bh(;x)}_{j-\lfloor |x|/2^i\rfloor})$ while if $j < \lfloor |x|/2^i \rfloor$, then $N_j^{i,x} = Ins_1(;\pm N_j^{i,Bh(;x)})$.

(2) For any node in $PATHS(x;)$ of height $i > 0$, the high bits of its two children are equal iff i is even.

(3) The following are equivalent, for any two leaves $N_{j_1}^{0,x}$ and $N_{j_2}^{0,x}$ in $PATHS(x;)$:

(a) The leaves have a common ancestor of height i.

(b) The leaves have the same rightmost $\lfloor |x|/2^i \rfloor$ bits.

(c) The leaves' deflations have the same rightmost $||x|| - i$ bits.

(4) If $PATHS(x;)$ is interpreted as a tree, its leaves comprise one full inflation of each path of length $||x||$.

Before proving this lemma formally, we illustrate it with the following

Example:

$PATHS(\lambda;) = \lambda,$ no blocks of length 0

$PATHS(1;) = \hat{0}\hat{1},$ 2 blocks of length 1

$PATHS(11;) = \overbrace{00}\,\overbrace{10}\,\overbrace{01}\,\overbrace{11},$ 4 blocks of length 2

$PATHS(1111;) = \overbrace{0000}\overbrace{1000}\overbrace{0010}\overbrace{1010}\overbrace{1111}\overbrace{0111}\overbrace{1101}\overbrace{0101},$

8 blocks of length 4, each of which is the full inflation of a distinct 3-bit path.

$PATHS(11111111;) =$

$$= \overbrace{00000000}\overbrace{10000000}\overbrace{00001000}\overbrace{10001000}\overbrace{10101010}\overbrace{00101010}\overbrace{10100010}\overbrace{00100010}$$
$$\overbrace{01010101}\overbrace{11010101}\overbrace{01011101}\overbrace{11011101}\overbrace{11111111}\overbrace{01111111}\overbrace{11110111}\overbrace{01110111},$$

16 blocks of length 8, each of which is the full inflation of a distinct 4-bit path.

The reader may also verify the remaining properties in these examples.

Proof of (1): Node $N_j^{i,x}$ is bits $j \cdot 2^i|x|$ through $(j+1)2^i|x| - 1$ of $PATHS(x;)$. If $j \geq \lfloor |x|/2^i \rfloor$, the node is in the front half of $PATHS(x;)$ and so it must be part of the result of $Ins_0(;PATHS(Bh(;x);))$. Specifically, by the definition of Ins_0, the portion of $PATHS(Bh(;x);)$ that affects the node in question is bits

$$\left\lfloor \left(j - \left\lfloor \frac{|x|}{2^i} \right\rfloor\right) \cdot 2^{i-1}|x| \right\rfloor \text{ through } \left\lfloor \left(j - \left\lfloor \frac{|x|}{2^i} \right\rfloor + 1\right) \cdot 2^{i-1}|x| \right\rfloor - 1$$

which is to say (since $|x| > 1$ is a power of 2)

$$\left(j - \left\lfloor \frac{|x|}{2^i} \right\rfloor\right) \cdot 2^i \left\lfloor \frac{|x|}{2} \right\rfloor \text{ through } \left(j - \left\lfloor \frac{|x|}{2^i} \right\rfloor + 1\right) \cdot 2^i \left\lfloor \frac{|x|}{2} \right\rfloor - 1$$

which are exactly the bits in node $N_{j-\lfloor |x|/2^i \rfloor}^{i,Bh(;x)}$, as desired.

If, however, $j < \lfloor |x|/2^i \rfloor$, then the node is in the back half of $PATHS(x;)$ so it must be part of the result of $Ins_1(;\pm PATHS(Bh(;x)))$; specifically, it is the result of applying Ins_1 and possibly Not to bits $j \cdot 2^i \lfloor |x|/2 \rfloor$ through $(j+1) \cdot 2^i \lfloor |x|/2 \rfloor - 1$ of $PATHS(Bh(;x))$, which are exactly the bits in node $N_j^{i,Bh(;x)}$.

Proof of (2): This time we do an induction on x. When $|x| = 1$, the only non-leaf node is 01, which has children with different high bits. For $|x| > 1$, consider node

$N_j^{i,x}$. Note that for any y,

$$\begin{aligned} HIBIT(;y) &= HIBIT(;Ins_0(;y)) \\ &= HIBIT(;Ins_1(;y)) \\ &\neq HIBIT(;Ins_1(;Not(;y))) \end{aligned}$$

If $i = ||x||$, then this is the root node, i.e. all of $PATHS(x;)$. Let v stand for $PATHS(Bh(;x);)$. The value of $PATHS(x;)$ is constructed from v as follows: if $HIBIT(;Fh(;v)) \neq HIBIT(;Bh(;v)))$, we concatenate $Ins_0(;v)$ and $Ins_1(;v)$, producing a result whose children have equal high bits; while if $HIBIT(;Fh(;v)) = HIBIT(;Bh(;v))$, we concatenate together $Ins_0(;v)$ and $Ins_1(;Not(;v))$, producing a result whose children have different high bits. In either case, whether the children's high bits are equal alternates from level to level.

If, on the other hand, $i < ||x||$, so we're not at the root node, then node $N_j^{i,x}$ is the result of an $Insert$ and possibly a Not on node $N_j^{i,Bh(;x)}$. By the induction hypothesis, the latter node has children with equal high bits iff i is even; but

$$\begin{aligned} HIBIT(;Fh(;N_j^{i,x})) &= HIBIT(;Fh(;Ins_?(;\pm N_j^{i,Bh(;x)}))) \\ &= \pm HIBIT(;Fh(;N_j^{i,Bh(;x)})) \end{aligned}$$

and

$$\begin{aligned} HIBIT(;Bh(;N_j^{i,x})) &= HIBIT(;Bh(;Ins_?(;\pm N_j^{i,Bh(;x)}))) \\ &= \pm HIBIT(;Bh(;N_j^{i,Bh(;x)})) \end{aligned}$$

so node $N_j^{i,x}$ too has children with equal high bits iff i is even.

Proof of (3): The (3b $\Rightarrow$ 3c) implication is obvious, since the rightmost $\lfloor |x|/2^i \rfloor$ bits of a leaf determine the rightmost $||x|| - i$ bits of its deflation.

We next prove the (3a $\Rightarrow$ 3b) implication, for all suitable x, by induction on i. When two leaves share an ancestor of height $i = 0$, they're actually the same leaf, and therefore share all $|x|/2^0$ of their bits. When $i = ||x||$, then the only node of height i is the root node, and *any* two leaves share their rightmost $\lfloor |x|/2^{||x||} \rfloor = 0$ bits. For $0 < i < ||x||$, consider distinct leaf nodes $N_{j_1}^{0,x}$ and $N_{j_2}^{0,x}$ with a common ancestor of height i, assuming wolog that $j_1 < j_2$. Leaves $N^{0,x}_{j_1}$ and $N^{0,x}_{j_2}$ must be in the same half of $PATHS(x;)$, since they have a common ancestor below the root. Suppose they're both in the back half, so $j_1 < j_2 < |x|$ (the case $|x| \leq j_1 < j_2$ is similar). Then both are formed by $Ins_1(;\pm\ldots)$ on leaves $N^{0,Bh(;x)}_{j_1}$ and $N^{0,Bh(;x)}_{j_2}$ respectively. The leaves $N^{0,Bh(;x)}_{j_1}$ and $N^{0,Bh(;x)}_{j_2}$ likewise have a common ancestor of height i, because the ancestor relation is independent of x. By the induction hypothesis, $N^{0,Bh(;x)}_{j_1}$ and $N^{0,Bh(;x)}_{j_2}$ share their rightmost $\left\lfloor \frac{|x|/2}{2^i} \right\rfloor$ bits. Applying Ins_1 or $Ins_1(;Not(;\ldots))$ to both of them produces results that share twice as many low bits, i.e. $\left\lfloor \frac{|x|}{2^i} \right\rfloor$ bits. (This last step doesn't work if $|x| < 2^i$ and hence $\left\lfloor \frac{|x|/2}{2^i} \right\rfloor = 0$, but in that case $\lfloor |x|/2^i \rfloor \leq 1$ so they need only share a single rightmost bit. This is automatic from the definitions of Ins_0 and Ins_1.)

The final direction, (3c $\Rightarrow$ 3a), is likewise proven by induction on i. Suppose $N^{0,x}_{j_1}$ and $N^{0,x}_{j_2}$ have distinct height-i ancestors $N^{i,x}_{k_1}$ and $N^{i,x}_{k_2}$, and suppose wolog that $k_1 < k_2$. Then

$$2^i k_1 \leq j_1 < 2^i(k_1+1) \leq 2^i k_2 \leq j_2 < 2^i(k_2+1).$$

If $j_1 < \left\lfloor \frac{|x|}{2^0} \right\rfloor \leq j_2$, then by (1) leaf $N^{0,x}_{j_1}$ is in the back half of $PATHS(x;)$ and is therefore the result of an Ins_1, while leaf $N^{0,x}_{j_2}$ is in the front half and is therefore the result of an Ins_0, so they differ in the rightmost bit, and hence in the rightmost bit of their deflations. Otherwise, assume $j_1 < j_2 < |x|$ (the case $|x| \leq j_1 < j_2$ is similar). As before, $N^{0,x}_{j_1}$ and $N^{0,x}_{j_2}$ are derived by $Insert$ and possibly Not operations from leaves $N^{0,Bh(;x)}_{j_1}$ and $N^{0,Bh(;x)}_{j_2}$. These are descendants of the (distinct) nodes $N^{i,Bh(;x)}_{k_1}$ and $N^{i,Bh(;x)}_{k_2}$, so by the induction hypothesis their deflations differ somewhere in the rightmost $|\frac{|x|}{2}| - i = ||x|| - i - 1$ bits. Applying Ins_0 or Ins_1 to either leaf has the effect of shifting its deflation left by one bit, filling with a 0 or 1 respectively, so the deflations of $N^{0,x}_{j_1}$ and $N^{0,x}_{j_2}$ differ somewhere in the rightmost $||x|| - i$ bits.

Proof of 4: Since there are $2|x| = 2^{||x||}$ leaves in all, it suffices to show that their deflations are all distinct, which follows immediately from the implication (3c $\Rightarrow$ 3a) above.

This concludes the proof of Lemma 16. ∎

Now that these general-purpose utility functions are defined, we turn our attention to the particular machine we wish to simulate. We assume without loss of generality that all computation paths in the machine are the same length, that the configurations of the machine form layers strictly alternating between universal and existential states (say, the last non-leaf layer is existential), that each non-leaf configuration has exactly two successor configurations, and that access to the input tapes is only at leaf configurations. These assumptions increase the alternating time by only a constant factor.

Suppose we're given an ALOGTIME machine M with j random-access input tapes and k work tapes, running in time bounded by $t = d{\cdot}\max(||\vec{X}||)$, where $X_1 \ldots X_j$ are the contents of the input tapes. Then

Definition 29 *A* configuration of M *is a tuple (in the sense of Definition 15)* $\langle q, IL_1, IR_1, \ldots IL_j, IR_j, WL_1, WR_1, \ldots WL_k, WR_k \rangle$ *where*

q *represents finite state information,*

IL_i *represents the contents of index tape i to the left of the head, with the lowest-significance bits representing the squares closest to the head,*

WL_i *the contents of work tape i to the left of the head (similarly),*

IR_i *the contents of index tape i from the head rightwards (lowest-significance bits representing the square under the head), and*

WR_i *the contents of work tape i from the head rightwards (similarly).*

Each element of the tuple is a string of length exactly t (assuming this is sufficient for the number of states).

Note that the inputs $\vec{X}$ are not an explicit part of a configuration. Note also that the lengths of the components are fixed and equal, so we can do simultaneous DCR on them.

But first we define the function $BLOWUP(\vec{X};)$ so as to yield a string of length $2^t - 1 = 2^{d \cdot \max(||\vec{X}||)} - 1 \doteq \max(|\vec{X}|)^d$, so that $||BLOWUP(\vec{X};)|| = d \cdot \max(||\vec{X}||) = t$, by composing d invocations of the $COPY$ function and suitable numbers of $Conc$ functions to deal with multiplicative constants.

Define the function $INIT\_TAPE(\vec{X};)$ to compute t from $|\vec{X}|$, then return the encoding of a blank tape of length t. Define $INIT\_Q(;)$ to return the encoding of the start state of the machine. Now for each component i of a machine configuration, define a function $INIT^i(\vec{X};)$ as either $INIT\_Q(;)$ (if $i = 1$) or $INIT\_TAPE(\vec{X};)$ (if $i > 1$); this is simply a notational convenience so we can easily use these functions as the base steps in a definition by simultaneous DCR.

For each component i of a machine configuration, we can define two functions $NEXT^i_L$ and $NEXT^i_R$, each taking a configuration C, to return the i-th components of the left and right successors, respectively, of the given configuration. Note that the current state can be tested bit-by-bit by a finite composition of $Cond$ and MSP functions, and a new state can be generated by concatenating constant functions. Similarly, the currently visible square of any tape can be extracted and tested with MSP and $Cond$, and tape head movement and writing can be easily done with MSP, $Conc$, and LSP. We have used no recursion in their definitions, so we can write them as $NEXT^i_L(;C)$ and $NEXT^i_R(;C)$.

From these functions $NEXT^i_L$ and $NEXT^i_R$ we define the $NEXT^i$ functions, which take a path P and use its high bit to decide whether to invoke $NEXT^i_L$ or $NEXT^i_R$.

$$NEXT(;P,C) \quad = \quad Cond(;HIBIT(;P), NEXT_L(;C), NEXT_R(;C))$$

Now we can use simultaneous, very safe DCR on the $NEXT^i$ functions to define

$$GETLEAF^i(P, \vec{X};) = $$
$$= \begin{cases} NEXT^i(; P, INIT^i(\vec{X};)) & \text{if } |P| \leq 1 \\ NEXT^i(; P, GETLEAF(Bh(; P), \vec{X};)) & \text{otherwise} \end{cases}$$

The $GETLEAF$ functions chop the inflated path in half repeatedly, each time extracting a single bit of path information (this is why the path was inflated in the first place) and finding a left or a right child according to that bit; if the path is empty, they use the starting configuration.

Next, having found our leaf configuration, we must decide whether it is an accepting configuration. This requires checking whether the state is an accepting state, a rejecting state, or a query state, and if the last, testing the appropriate bit of the appropriate input (using $GETBIT$ on the "normal" input parameters $\vec{X}$). No recursion is required except that to define $GETBIT$, and we can define $EVALLEAF(\vec{X}; C)$ to output a 1 or a 0 depending on whether the configuration C, with input $\vec{X}$, accepts or not.

Now we can define the $\{0, 1\}$-valued function $EVALTREE(T, s, \vec{X}; C)$ as follows (s is intended to be any fully-inflated path, i.e. $|s| = 2^{t-1}$):

$$EVALTREE(T, s, \vec{X};) = $$
$$= \begin{cases} EVALLEAF(\vec{X}; GETLEAF(T, \vec{X};)) \\ \quad \text{if } |T| \leq |s| \\ EVALTREE(Fh(; T), s, \vec{X};) \wedge EVALTREE(Bh(; T), s, \vec{X};) \\ \quad \text{if } |T| > |s| \text{ and the height of } T \text{ is odd} \\ EVALTREE(Fh(; T), s, \vec{X};) \vee EVALTREE(Bh(; T), s, \vec{X};) \\ \quad \text{if } |T| > |s| \text{ and the height of } T \text{ is even} \end{cases}$$

or, more formally,

$$
\begin{array}{l}
EVALTREE(T,s,\vec{X};) = \\
\quad = \left\{ \begin{array}{l}
EVALLEAF(\vec{X}; GETLEAF(T,\vec{X};)) \\
\quad \text{if } |T| \le |s| \\
h(;T, EVALTREE(Fh(;T),s,\vec{X};), EVALTREE(Bh(;T),s,\vec{X};)) \\
\quad \text{if } |T| > |s|
\end{array} \right.
\end{array}
$$

where

$$
\begin{array}{rcl}
h(;T,u_1,u_2) & = & Cond(;HIBIT(;T), \\
& & \qquad Cond(;HIBIT(;Bh(;T)), \\
& & \qquad\qquad Cond(;u_1,1,u_2), \\
& & \qquad\qquad Cond(;u_1,u_2,0)), \\
& & \qquad Cond(;HIBIT(;Bh(;T)), \\
& & \qquad\qquad Cond(;u_1,u_2,0), \\
& & \qquad\qquad Cond(;u_1,1,u_2)))
\end{array}
$$

Finally, $|BLOWUP(\vec{X};)| = 2^t - 1$, so $|Bh(;BLOWUP(\vec{X};))| = \left\lceil \frac{2^t-1}{2} \right\rceil = 2^{t-1}$, so the simulation of the ATM can be accomplished by

$$
\begin{array}{rcl}
SIMULATE(\vec{X};) & = & EVALTREE(PATHS(BLOWUP(\vec{X};);), \\
& & \qquad Bh(;BLOWUP(\vec{X};)), \vec{X};)
\end{array}
$$

Thus $ALOGTIME \subseteq vsc(BASE)$. This completes the proof of Lemma 15.

■

3.5 Parallelism and Comprehension

Since the above lemma shows $vsc(BASE)$ can simulate any ALOGTIME machine, many other useful functions immediately fall into $vsc(BASE)$, indirectly by use of the PATHS-tree technique. But the technique is useful in its own right for direct definition of useful functions, not just for simulating Turing machines. For example, the $\{0,1\}$-valued function that indicates whether two given, equal-length bit-strings are equal is in FO, hence certainly in ALOGTIME, and hence in $vsc(BASE)$; but we can define it explicitly without resorting to Turing machines as follows: given equal-length strings x and y, let

$$\begin{aligned}
EQUAL(x,y;) &= AUX(PATHS(x;), EXTEND(x;), EXTEND(y;);) \\
AUX(T,x,y;) &= \begin{cases} AUX(Fh(;T),x,y) \wedge AUX(Bh(;T),x,y) & \text{if } |T| > |x| \\ EQBIT(CHOP(T,x;), CHOP(T,y;);) & \text{if } |T| \le |x| \end{cases} \\
EQBIT(b_1,b_2;) &= \begin{cases} 0 & \text{if bit } b_1 \text{ equals bit } b_2 \\ 1 & \text{if not, or if } |b_1| \ne 1 \text{ or } |b_2| \ne 1 \end{cases} \\
CHOP(T,x;) &= \begin{cases} Bh(;CHOP(Bh(;T),x;)) & \text{if } |T| > 1 \text{ and } HIBIT(;T) = 1 \\ Fh(;CHOP(Bh(;T),x;)) & \text{if } |T| > 1 \text{ and } HIBIT(;T) = 0 \\ x & \text{if } |T| \le 1 \end{cases}
\end{aligned}$$

The $CHOP$ function follows the inflated path T to extract the single bit specified by reading it as a sequence of Fh's and Bh's. The $EQBIT$ function then compares the two bits thus extracted from strings x and y, and AUX computes the conjunction of $EQBIT$ over all the possible paths, thus indicating whether all the bits of x are equal to the corresponding bits of y.

The above definition of $EQUAL$ makes no use of property 2 of the PATHS function; accordingly, we could define a simplified PATHS function

$$\begin{aligned}
EZPATHS(x;) &= \begin{cases} \lambda & \text{if } |x| = 0 \\ 01 & \text{if } |x| = 1 \\ h(;EZPATHS(Bh(;x);)) & \text{if } |x| > 1 \end{cases} \\
h(;v) &= Conc(;Ins_0(;v), Ins_1(;v))
\end{aligned}$$

which retains properties 1, 3, and 4 of PATHS but which lists the inflated paths in a more natural, colexicographic order.

By replacing $PATHS$ with $EZPATHS$, and also replacing the $\wedge$ in the definition of AUX above with a $Conc$, we could define, given any function $g(b_1, b_2, \ldots b_k)$ and equal-length strings $x_1, x_2, \ldots x_k$, the concatenation, for $0 \le i < |x_1|$, of the values

$g(Bit(i, x_1), Bit(i, x_2), \ldots Bit(i, x_k))$, effectively computing the function g in parallel for each bit of input. This suggests the technique we shall use for comprehension in Lemma 17.

Lemma 15 demonstrated that any ALOGTIME relation is in $vsc(BASE)$, but we have yet to show that ALOGTIME *functions* (by which we mean, as usual, functions whose bit-graphs are in ALOGTIME) are in $vsc(BASE)$. The following lemma remedies this.

Lemma 17 *For any function $F(\vec{x};)$, if there are functions $f(\vec{x}, i;)$ and $SIZE_F(\vec{x};)$ in $vsc(BASE)$ such that*

- *for all $\vec{x}$, $|SIZE_F(\vec{x};)|$ is a power of 2, and*
- *for all $\vec{x}$, $|F(\vec{x};)| = |SIZE_F(\vec{x};)|$, and*
- *for all $\vec{x}$ and all $i < |F(\vec{x};)|$ we have $f(\vec{x}, i;) = Bit(i, F(\vec{x};))$,*

then $F(\vec{x};) \in vsc(BASE)$.

Exactly the same holds with $sc(BASE)$ in place of $vsc(BASE)$.

Proof of Lemma 17: We build a binary tree, similar to $PATHS$, each of whose leaves contains a different value, but this time the values are the binary numbers $0 \ldots |SIZE_F(\vec{x};)| - 1$ in reverse order. We can then define F by very safe DCR on such a tree T as follows:

$$F(\vec{x};) = F'(T, Len(SIZE_F(\vec{x};);), \vec{x}; \vec{y})$$

$$F'(T, z, \vec{x};) = \begin{cases} f(T, \vec{x};) & \text{if } |T| \leq |z| \\ Conc(; F'(Fh(;T), z, \vec{x};), F'(Bh(;T), z, \vec{x};)) & \text{if } |T| > |z| \end{cases}$$

The iterated function is $Conc$, which is associative; so the effect is simply to concatenate the bits $f(|z| - 1, \vec{x};)$, $f(|z| - 2, \vec{x};), \ldots f(1, \vec{x};)$, $f(0, \vec{x};)$, where the numbers $|z| - 1, |z| - 2, \ldots 1, 0$ are given to f in binary. (Note that if F were allowed safe parameters, and they affected its length, $SIZE_F$ would depend on those parameters, and therefore could not be used in a normal position in F'. This is why the F in this lemma only takes normal parameters; the lemma is somewhat weaker than we might like, but it suffices to prove Theorem 9.)

All that remains is to build, given a string z whose length is a power of 2, a list of the binary numbers from $0 \ldots |z|-1$ in reverse order. We define it by simultaneous, very safe DCR:

$$COUNTUP(z;) = f_2(z; Len(z;))$$

$$f_1(z;b) = \begin{cases} SHL(;ZEROES(;b),1) & \text{if } |z| \leq 1 \\ Conc(;SHL(;f_1(Bh(;z);b),0), SHL(;f_1(Bh(;z);b),0)) & \text{if } |z| > 1 \end{cases}$$

$$f_2(z;b) = \begin{cases} ZEROES(;b) & \text{if } |z| \leq 1 \\ Conc(;Or(;f_1(Bh(;z);b), f_2(Bh(;z);b)), f_2(Bh(;z);b)) & \text{if } |z| > 1 \end{cases}$$

Clearly, $|f_1| = |f_2| = |b|$ at the base level, and each of f_1 and f_2 exactly doubles in length at each recursive step, so this definition meets the requirements of Lemma 5, and so f_1, f_2, and $COUNTUP$ are within $vsc(BASE)$.

Whenever $|z| \geq 1$, $f_1(z;b)$ is the concatenation of $2^{\lceil \log(|z|) \rceil}$ identical blocks of length $|b|$, each composed of zeroes except for a 1 in bit position $\lceil \log(|z|) \rceil$: shifting such a list of blocks left by one bit has the effect of shifting over the 1 in each block simultaneously. As for f_2, the value of $f_2(z;b)$ starts as a block of $|b|$ zeroes (for $|z|=1$), and at each subsequent step two copies of it are concatenated, the first having a high bit set in each of its blocks by OR-ing with $f_1(z;b)$. Thus if $f_2(Bh(;z);b)$ lists the binary numbers $2^{\lceil \log(|z|) \rceil - 1} - 1, \ldots 1, 0$ in blocks of size $|b|$, then $f_2(z;b)$ lists the numbers $2^{\lceil \log(|z|) \rceil} - 1, \ldots 2^{\lceil \log(|z|) \rceil - 1}$ followed by $2^{\lceil \log(|z|) \rceil - 1} - 1, \ldots 1, 0$, so the whole list is $2^{\lceil \log(|z|) \rceil} - 1, \ldots 1, 0$ as desired. Plugging the resulting $COUNTUP(z;)$ in place of the T in the above definition of F completes the proof of the lemma. ∎

Corollary 18 *If $F(\vec{x};)$ is any function whose length is exactly a polynomial (with nonnegative integer coefficients) in the lengths of the inputs, and if the characteristic function of the bit-graph of F is in $vsc(BASE)$, then $F \in vsc(BASE)$.*

Proof of Corollary: Any polynomial growth rate with nonnegative integer coefficients can be attained within $vsc(BASE)$ by suitable composition of $COPY$ and $Conc$ functions; let $g(\vec{x};)$ be such a function. Then $ROUNDUP(g(\vec{x};);)$ has length a power of 2, and is at least as long as g, so we can use it for $SIZE_F$. The bit-graph of F is the set of tuples $\langle \vec{x}, i, c \rangle$ such that bit i of $F(\vec{x};)$ is equal to c, so the function $f(\vec{x}, i;)$ of Lemma 17 is simply the characteristic function of the bit-graph, with $c = 1$, and is therefore in $vsc(BASE)$. We don't care what this function produces for values of i greater than $|g(\vec{x};)|$; the proof of Lemma 17 still produces a function F' whose low $|g(\vec{x};)|$ bits are those of F. Thus $LSP(;F'(\vec{x};), g(\vec{x};)) = F(\vec{x};)$, and this function is in $vsc(BASE)$. ∎

This comprehension capability allows us to pass from a function defined by its bit-graph (as produced by Lemma 15) to the function itself. In combination with Lemmas 10 and 12, this completes the proof of Theorem 9. ∎

Chapter 4

Characterizing Polylog-Depth Circuits

The natural growth rates of functions in this framework seem to be exponential in circuit depth. When we investigate circuits whose depth is polynomial in the logarithm of input size, the corresponding growth rates are $2^{poly(||\vec{x}||)}$. The main theorem of the present chapter is

Theorem 19 *For any smooth function $f(\vec{x};)$, the following are equivalent:*

(1) $f(\vec{x};) \in sc(BASE)$

(2) $f(\vec{x})$ *can be computed from* $\vec{x}$ *by a* U_E*-uniform, polylog depth circuit family, and* f *has* $2^{poly(||\vec{x}||)}$ *growth rate*

(3) the i-th bit of $f(\vec{x})$ *can be computed from* i *and* $\vec{x}$ *in alternating polylog time, and* f *has* $2^{poly(||\vec{x}||)}$ *growth rate*

(4) $f(\vec{x})$ *can be computed by a deterministic Turing machine in polylogarithmic space (not counting a write-only output tape)*

The proof is similar to that of Theorem 9. We prove $(1 \Rightarrow 2)$ and $(3 \Rightarrow 1)$ directly. The equivalence $(2 \Leftrightarrow 3)$ follows from Theorems 3 and 4 (due to Ruzzo

[Ruz81]). The ($4 \Rightarrow 3$) implication follows from Theorem 3.1 of [CKS81], and ($3 \Rightarrow 4$) follows from a result of Borodin, quoted in [CKS81] as Theorem 4.2. Note that part 2 of the theorem uses U_E uniformity; the theorem also holds with U_E replaced by U_D, U_{E^*}, or $U_{E'}$, since for polylog-depth, $2^{polylog}$-size circuits all four are equivalent, but U_E turns out most convenient for our purposes.

Lemma 20 *If $f(\vec{x};\vec{y}) \in sc(BASE)$, then there is a polynomial p_f with nonnegative coefficients such that*

$$|f(\vec{x};\vec{y})| \leq 2^{p_f(||\vec{x}||)} \cdot \max(|\vec{y}|, 1)$$

Proof of Lemma 20: As pointed out before, all the BASE functions have linear growth rate, easily satisfying this criterion.

Definition by Safe Composition: If $f(\vec{x};\vec{y})$ is defined by safe composition as $g(u(\vec{x}); v(\vec{x};\vec{y}))$, and if p_g, p_u, and p_v are polynomials satisfying the Lemma for functions g, u, and v respectively, then

$$\begin{aligned} |f(\vec{x};\vec{y})| &\leq 2^{p_g(||u(\vec{x})||)} \cdot |v(\vec{x};\vec{y})| \\ &\leq 2^{p_g(|2^{p_u(||\vec{x}||)}|)} \cdot 2^{p_v(||\vec{x}||)} \cdot \max(|\vec{y}|, 1) \\ &\leq 2^{p_g(p_u(||\vec{x}||)+1)} \cdot 2^{p_v(||\vec{x}||)} \cdot \max(|\vec{y}|, 1) \\ &\leq 2^{p_g(p_u(||\vec{x}||)+1)+p_v(||\vec{x}||)} \cdot \max(|\vec{y}|, 1) \end{aligned}$$

so the statement is satisfied by letting

$$p_f(||\vec{x}||) = p_g(p_u(||\vec{x}||) + 1) + p_v(||\vec{x}||)$$

Definition by Safe DCR: On the other hand, if $f(z, b, \vec{x};\vec{y})$ is defined by safe DCR on the parameter z, to wit:

$$f(z, b, \vec{x};\vec{y}) = \\ = \begin{cases} g(z, \vec{x};\vec{y}) & \text{if } |z| \leq \max(|b|, 1) \\ h(z, \vec{x};\vec{y}, f(Fh(;z), b, \vec{x};\vec{y}), f(Bh(;z), b, \vec{x};\vec{y})) & \text{if } |z| > \max(|b|, 1) \end{cases}$$

and p_g and p_h are polynomials satisfying the Lemma for functions g and h respectively, then we prove by induction on $|z|$ that

$$|f(z,b,\vec{x};\vec{y})| \quad \leq \quad 2^{\lceil \log(|z|)\rceil \cdot p_h(||z||,||\vec{x}||)+p_g(||b||,||\vec{x}||)} \cdot \max(|\vec{y}|,1)$$

The base case, for $|z| \leq |b|$, is obvious. For larger $|z|$, we know

$$\begin{aligned}
|f(z,b,\vec{x};\vec{y})| &\leq 2^{||z||,||\vec{x}||)} \cdot \max(|f(Fh(;z),b,\vec{x};\vec{y})|,|f(Bh(;z),b,\vec{x};\vec{y})|,|\vec{y}|) \\
&\leq 2^{p_h(||z||,||\vec{x}||)} \cdot \max\left(2^{\lceil \log\left(\lceil \frac{|z|}{2}\rceil\right)\rceil \cdot p_h\left(|\lceil \frac{|z|}{2}\rceil|,||\vec{x}||\right)+p_g(||b||,||\vec{x}||)}, |\vec{y}|\right) \\
&\leq 2^{p_h(||z||,||\vec{x}||)} \cdot \max\left(2^{(\lceil \log(|z|)\rceil -1)\cdot p_h(||z||,||\vec{x}||)+p_g(||b||,||\vec{x}||)}, |\vec{y}|\right) \\
&\leq 2^{p_h(||z||,||\vec{x}||)} \cdot 2^{(\lceil \log(|z|)\rceil -1)\cdot p_h(||z||,||\vec{x}||)+p_g(||b||,||\vec{x}||)} \cdot \max(|\vec{y}|) \\
&\leq 2^{\lceil \log(|z|)\rceil \cdot p_h(||z||,||\vec{x}||)+p_g(||b||,||\vec{x}||)} \cdot \max(|\vec{y}|)
\end{aligned}$$

as desired. ∎

Lemma 21 *If $f(\vec{x};\vec{y}) \in sc(BASE)$, then there is a polynomial q_f with nonnegative coefficients and a U_{E^*}-uniform circuit family F computing f's bit-graph such that for all $\vec{x},\vec{y}$,*

$$d_F[\vec{x};\vec{y}] \leq q_f(||\vec{x}||) \cdot \max(||\vec{y}||,1)$$

Proof of Lemma 21: All the BASE functions can obviously be computed in such a fashion.

Definition by Safe Composition: If $f(\vec{x};\vec{y})$ is defined by safe composition as $g(u(\vec{x};);v(\vec{x};\vec{y}))$, then let p_g, p_u, and p_v be polynomials whose exponentials bound the growth of g, u, and v respectively as in Lemma 20, and let q_g, q_u, and q_v be polynomials bounding the circuit depth required to compute g, u, and v with circuit families G, U, and V respectively, by an inductive hypothesis on the present lemma. And let F be the obvious circuits for f: a copy of circuit G, with each input replaced by a copy of circuit U or V as appropriate.

$$\begin{aligned}
&d_F[\vec{x};\vec{y}] \leq \\
&\quad \leq \quad d_G[u(\vec{x};);v(\vec{x};\vec{y})] + \max(d_U[\vec{x};],d_V[\vec{x};\vec{y}])
\end{aligned}$$

$$
\begin{aligned}
&\leq q_g(||u(\vec{x};)||) \cdot ||v(\vec{x};\vec{y})|| + \max(q_u(||\vec{x}||), q_v(||\vec{x}||) \cdot \max(||\vec{y}||, 1)) \\
&\leq q_g(p_u(||\vec{x}||)) \cdot \max(p_v(||\vec{x}||) + \max(||\vec{y}||, 1)) + \\
&\quad \max(q_u(||\vec{x}||), q_v(||\vec{x}||) \cdot \max(||\vec{y}||, 1)) \\
&\leq q_g(p_u(||\vec{x}||)) \cdot p_v(||\vec{x}||) \cdot \max(||\vec{y}||, 1) + (q_u(||\vec{x}||) + q_v(||\vec{x}||)) \cdot \max(||\vec{y}||, 1) \\
&= (q_g(p_u(||\vec{x}||)) \cdot p_v(||\vec{x}||) + q_u(||\vec{x}||) + q_v(||\vec{x}||)) \cdot \max(||\vec{y}||, 1)
\end{aligned}
$$

so the statement is satisfied by letting

$$q_f(\vec{r}) = q_g(p_u(\vec{r})) \cdot p_v(\vec{r}) + q_u(\vec{r}) + q_v(\vec{r})).$$

The uniformity of the resulting circuits is by the same method as in Chapter 3, though there are several minor differences. First, the circuits are now $2^{polylog}$ in size and so the algorithm needed for $U_{E'}$- or U_{E^*}-uniformity has alternating polylog time to run. Second, since we state the theorem in terms of U_E-uniformity, we must invoke Theorem 2 to draw the desired conclusion. And third, in this setting we need not worry about mapping-uniformity. Again, the proof in the case that there are multiple functions $\vec{u}$ and $\vec{v}$ is similar.

Definition by Safe DCR: Now suppose $f(z, b, \vec{x}; \vec{y})$ is defined by safe DCR on the parameter z, to wit:

$$
\begin{aligned}
&f(z, b, \vec{x}; \vec{y}) = \\
&= \begin{cases} g(z, \vec{x}; \vec{y}) & \text{if } |z| \leq \max(|b|, 1) \\ h(z, \vec{x}; \vec{y}, f(Fh(;z), b, \vec{x}; \vec{y}), f(Bh(;z), b, \vec{x}; \vec{y})) & \text{if } |z| > \max(|b|, 1) \end{cases}
\end{aligned}
$$

y) Let p_f, p_g, and p_h be polynomials satisfying Lemma 20 for functions f, g, and h respectively, and q_g and q_h be polynomials satisfying Lemma 21 for circuit families G and H. And let F be the obvious circuits for f: an instance of circuit H, but replacing each of its inputs by a smaller instance of F or an instance of G, depending on whether $|z| > |b|$.

We know that, for all $|z'| \leq |z|$,

$$|f(z', \vec{x}; \vec{y})| \leq 2^{p_f(||z||, ||b||, ||\vec{x}||)} \cdot \max(|\vec{y}|, 1)$$

so if $h(z, \vec{x}; u_1, u_2, \vec{y})$ is called in the course of computing $f(z, b, \vec{x}; \vec{y})$, we can be assured that

$$|u_1|, |u_2| \leq 2^{p_f(||z||,||b||,||\vec{x}||)} \cdot \max(|\vec{y}|, 1)$$

and therefore $||u_1||$ and $||u_2||$ are each at most $p_f(||z||, ||b||, ||\vec{x}||) + \max(||\vec{y}||, 1)$. With this in mind, then,

$$\begin{aligned}
& d_F[z, b, \vec{x}; \vec{y}] \leq \\
& \quad \leq \max(\ d_G[z, \vec{x}; \vec{y}], \\
& \qquad d_H[z, \vec{x}; f(Fh(;z), b, \vec{x}; \vec{y}), f(Bh(;z), b, \vec{x}; \vec{y})] + \\
& \qquad \max(d_F[Fh(;z), b, \vec{x}; \vec{y}], d_F[Bh(;z), b, \vec{x}; \vec{y}])) \\
& \quad \leq \max(d_G[z, \vec{x}; \vec{y}], \\
& \qquad d_H[z, \vec{x}; f(Fh(;z), b, \vec{x}; \vec{y}), f(Bh(;z), b, \vec{x}; \vec{y}), \vec{y}] + \\
& \qquad d_F[Bh(;z), b, \vec{x}; \vec{y}]) \\
& \qquad \text{by monotonicity of } d_F \\
& \quad \leq d_G[z, \vec{x}; \vec{y}] + \\
& \qquad \left(\sum_{i=||b||+1}^{||z||} q_h(i, ||\vec{x}||)\right) \cdot \\
& \qquad \max(p_f(||z||, ||b||, ||\vec{x}||) + \max(||\vec{y}||, 1), ||\vec{y}||) \\
& \qquad \text{by the remarks above} \\
& \quad \leq d_G[z, \vec{x}; \vec{y}] + ||z|| \cdot q_h(||z||, ||\vec{x}||) \cdot \\
& \qquad (p_f(||z||, ||b||, ||\vec{x}||) + \max(||\vec{y}||, 1)) \\
& \qquad \text{by monotonicity of } q_h \\
& \quad \leq q_g(||z||, ||\vec{x}||) \cdot \max(||\vec{y}||, 1) + ||z|| \cdot q_h(||z||, ||\vec{x}||) \cdot \\
& \qquad (p_f(||z||, ||b||, ||\vec{x}||) + \max(||\vec{y}||, 1)) \\
& \quad \leq (||z|| \cdot q_h(||z||, ||\vec{x}||) \cdot (p_f(||z||, ||b||, ||\vec{x}||) + 1) + \\
& \qquad q_g(||z||, ||\vec{x}||)) \cdot \max(||\vec{y}||, 1)
\end{aligned}$$

and so it would suffice to let $q_f(||z||, ||b||, ||\vec{x}||)$ be

$$||z|| \cdot q_h(||z||, ||\vec{x}||) \cdot (p_f(||z||, ||b||, ||\vec{x}||) + 1) + q_g(||z||, ||\vec{x}||)$$

∎

Thus each bit of $f(\vec{x};\vec{y})$ can be computed in polylog depth, $2^{polylog}$ size. Since there are at most $2^{polylog}$ bits of output, the whole function value is computable by polylog-depth, $2^{polylog}$-size circuits.

As for uniformity, we use the same gate-numbering scheme as in the corresponding proof in Chapter 3. We no longer know that the functions we're iterating are constant-depth, much less mapping-uniform, but we don't need to, since by Theorem 1 we need only recognize the *direct* connection language in deterministic polylog time, and the extended connection language will come as a corollary. By hypothesis the circuit families G and H are U_E-uniform, so in constructing this deterministic polylog-time algorithm we can invoke deterministic polylog-time algorithms for the extended connection languages of G and H. The direct connection language requires that we recognize gate types (which is done by simply invoking the uniformity algorithms for G or H) and *direct* descendants. To tell whether gate a is, say, the left child of gate b, first check whether they descend from the same output bit; if not, answer "no". If b is not an input gate to its local H or G circuit, check that the $\{Fh, Bh\}$ paths to a and b are identical (if not, answer "no") and invoke the H or G algorithm to see whether a is the left child of b. If, on the other hand, b *is* an input gate to the H or G circuit of which it's a part, use the H or G uniformity algorithm to find out which input bit of which parameter it is, and make sure a is the correct output bit of the correct parameter. ∎

This is one direction of Theorem 19. For the remaining direction, we shall show that

Lemma 22 *Given an alternating Turing machine M with inputs $\vec{X}$ and time bound polynomial in $\log(|\vec{X}|)$, there is a function $f \in sc(BASE)$ such that*

$$f(\vec{X};) = \begin{cases} 1 & \textit{if } M \textit{ accepts } \vec{X} \\ 0 & \textit{if } M \textit{ rejects } \vec{X} \end{cases}$$

In other words, $sc(BASE)$ can simulate the computation of an arbitrary polylog-time alternating machine.

The proof of this is almost identical to that of Lemma 15. The difference is that with merely-safe DCR we can define not only $COPY$ but $\#_3$, and with this growth rate we can build and evaluate a binary tree of polylog rather than merely $O(\log)$ depth.

With $\#_3$ in hand, then, and given an alternating machine with time bound $t = p_M(||\vec{X}||)$, we can redefine $BLOWUP(\vec{X};)$ to have length $2^{p_M(||\vec{X}||)} - 1$, and hence $||BLOWUP(\vec{X};)|| \geq p_M(||\vec{X}||)$, by composing $\deg(p_M)$ invocations of the $\#_3$ function as well as suitable numbers of $COPY$ and $Conc$ functions.

We define the functions $ZEROES$, $INIT_M$, $GETBIT$, $NEXT_L$, $NEXT_R$, $NEXT$, $PATHS$, $EVALLEAF$, $GETLEAF$, $EVALTREE$, and $SIMULATE$ exactly as before; the result is that $EVALTREE$ now evaluates an alternating and/or tree of depth $p_M(||\vec{X}||)$ rather than $d \cdot \max(||\vec{X}||)$, and thus correctly simulates the alternating polylog-time Turing machine.

Lemma 23 *If $F(\vec{x};)$ is any function whose length is exactly $2^{p(||\vec{x}||)}$ for p a polynomial with nonnegative integer coefficients, and if the characteristic function of the bit-graph of F is in $sc(BASE)$, then $F \in sc(BASE)$.*

Proved exactly as Corollary 18, with larger growth rates.

Chapter 5

Conclusions and Observations

5.1 Concatenation Recursion

The most unfamiliar functions in BASE are, of course, Ins_0 and Ins_1, although they are in FNC^0 and hence not computationally difficult. An alternate way to prove the theorems in this paper would be to omit them but add some form of concatenation recursion, analogous to the Concatenation Recursion on Notation introduced in [Clo89b] to help characterize AC^0.

Definition 30 (Clote) *A function $f(z, \vec{x})$ on natural numbers is defined by* concatenation recursion on notation *from functions $h_0(z, \vec{x})$, $h_1(z, \vec{x})$, and $g(\vec{x})$ if h_0 and h_1 are $\{0, 1\}$-valued functions and*

$$
\begin{aligned}
f(0, \vec{x}) &= g(\vec{x}) \\
f(z, \vec{x}) &= \begin{cases} 2 \cdot f(\lfloor \frac{z}{2} \rfloor, \vec{x}) + h_0(z, \vec{x}) & \text{if } z > 0 \text{ is even} \\ 2 \cdot f(\lfloor \frac{z}{2} \rfloor, \vec{x}) + h_1(z, \vec{x}) & \text{if } z \text{ is odd} \end{cases}
\end{aligned}
$$

According to this definition, the value of f is the concatenation of $g(\vec{x})$ with $|z|$ bits each computed independently from a corresponding prefix of z. In some applications the prefix of z is essential, for example in computing the conjunction or disjunction of the bits of z. In other applications, such as computing the equivalent of our *Not* function, it would suffice for h to be given only $\vec{x}$, or $\vec{x}$ and a constant number of bits of z. The distinction corresponds to that between unbounded-fanin and bounded-fanin gates, as the latter, more restricted form of CRN would allow each bit of output to depend on only constantly many bits of input.

As this paper treats bounded-fanin more than unbounded-fanin circuits, we define forms of recursion which are more restricted in this sense. But on the other hand, we want to be able to easily generate functions with more than one bit of output naturally corresponding to each bit of input, so we replace Clote's restriction of each h function to a single bit of output by a restriction to a constant, power-of-2 number of bits of output. The base function g of Clote's definition can be omitted in our setting, since we have the concatenation function as a primitive operation anyway.

Definition 31 *A function $f(\vec{x}; z, \vec{y})$ on bit-strings is said to be defined by k-*bit concatenation recursion on notation *(or k-*CRN*) from function $h(\vec{x}; z, \vec{y})$ if k is a fixed power of 2, $|h(\vec{x}; z, \vec{y})| = k$ for all $\vec{x}, \vec{y}, z$, and*

$$f(\vec{x}; z, \vec{y}) = \begin{cases} \lambda & \text{if } |z| = 0 \\ Conc(; f(\vec{x}; DELLOBIT(; z), \vec{y}), & \text{if } |z| > 0 \\ \qquad h(\vec{x}; LOBIT(; z), \vec{y})) & \end{cases}$$

Or, if we prefer the new scheme to resemble DCR more closely, we can use instead the following formulation:

Definition 32 *A function $f(\vec{x}; z, \vec{y})$ is said to be defined by k-*bit concatenation DCR *(or k-*CDCR*) from function $h(\vec{x}; z, \vec{y})$ if k is a fixed power of 2, $|h(\vec{x}; z, \vec{y})| = k$ for all $\vec{x}, \vec{y}, z$, and*

$$f(\vec{x}; z, \vec{y}) = \begin{cases} \lambda & \text{if } |z| = 0 \\ h(\vec{x}; z, \vec{y}) & \text{if } |z| = 1 \\ Conc(; f(\vec{x}; Fh(; z), \vec{y}), & \text{if } |z| > 1 \\ \qquad f(\vec{x}; Bh(; z), \vec{y})) & \end{cases}$$

In either case, $f(\vec{x}; z, \vec{y})$ is the concatenation of all $|z|$ of the k-bit strings produced by h given the individual bits of z. In passing from safe DCR to k-CDCR or k-CRN we have gained the ability to recurse on a safe parameter, but paid for it by restricting the leaf values to have a fixed length and allowing only $Conc$ as an iterated function.

For example, we could define Ins_0 as follows by 2-CDCR:

$$Ins_0(; x) \quad = \quad \begin{cases} \lambda & \text{if } |x| = 0 \\ Conc(; x, 0) & \text{if } |x| = 1 \\ Conc(; Ins_0(; Fh(; x)), Ins_0(; Bh(; x))) & \text{otherwise} \end{cases}$$

Clearly $|f(\vec{x}; z, \vec{y})| = |z| \cdot k$, so k-CDCR and k-CRN produce growth rates no more than linear in the lengths of safe parameters. Furthermore, they have no effect whatsoever on the depth of ALOGTIME-uniform circuits, as to find a desired bit position i of the output one need only divide i by k and extract bit $i \bmod k$ of $h(\vec{x}; Bit(\lfloor i/k \rfloor, z), \vec{y})$. This can all be done within the ALOGTIME (indeed, DLOGTIME) uniformity condition, since k is a power of 2. For similar reasons, k-CDCR and k-CRN add only $O(\log)$ to the run-time of alternating Turing machines. Note that although the parameter of recursion is allowed to be safe, it does not affect the *depth* of circuits, only their breadths.

The result is that Theorems 9 and 19 would still hold if the BASE functions Ins_0 and Ins_1 were deleted and the 2-CDCR or 2-CRN recursion scheme added in their place.

5.2 Lengths of Circuit Functions

The observant reader will notice a peculiar stipulation in Corollary 18: that the function F being built by comprehension have not merely polynomial-bounded length, but *exactly* polynomial length. Since $vsc(BASE)$ and $sc(BASE)$ were carefully defined so that the output lengths of their functions are determined by the lengths of their inputs, they obviously cannot contain any function whose output length is *not* a function of the lengths of its inputs. Furthermore, there may be other functions excluded from $vsc(BASE)$ and/or $sc(BASE)$ because their output lengths are *computationally difficult* functions of their input lengths. Of course, such functions (so long as they have suitably *bounded* growth) can always be padded with zeroes to a suitable length, and Theorems 9 and 19 apply perfectly well to the padded versions.

5.3 Related Work in the Field

Our Theorem 9 bears a resemblance to Theorem 3.3 of [CL90], which first defines the "simple functions" as the closure of a few specific functions (similar to our BASE functions) under composition, and then characterizes ALOGTIME as the closure of the simple functions, together with an equality test, under composition and "upward tree recursion", a scheme that allows branching iteration of any simple function. Our analogue of Compton & LaFlamme's simple functions are all those functions with no normal parameters, a purely syntactic definition which we feel makes the recursion scheme's definition cleaner. In addition, we need no *a priori* polynomial size bound, need not treat the equality test as a special case, and our bit-string formulation seems more natural than the finite-models formulation in that paper.

Theorem 19 can be seen as analogous to Theorem 2.1 of [All91], which characterizes NC, circuits of simultaneous polylog depth and polynomial size. The reason Allen's system describes NC while Theorem 19 describes polylog-depth circuits is simply that Allen's recursion schemes involve an explicit polynomial size bound. ([All91] actually gives both an recursion-theoretic and a proof-theoretic characterization of NC, the latter using the theory D^1, which is equivalent to the bounded arithmetic theory R^1_2. Buss, Krajíček, and Takeuti [BKT92], by replacing R^1_2 with R^1_3 (whose language allows terms of $2^{polylog}$ growth rate), have characterized polylog space prooftheoretically.) The present result also requires only two function-combination schemes (safe composition and safe DCR) rather than the four (composition, p-comprehension, log-bounded recursion on notation, and polynomial-bounded branching recursion) in [All91], and uses simpler base functions. Alternatively, we can add the CDCR or CRN scheme described above and make our base functions even simpler and more natural.

5.4 Directions for Future Research

We have extended the technique of distinguishing safe parameters, pioneered in [BC92], to characterize both polylog space and alternating log time, and demonstrated its power and generality. We speculate that further variants on this technique may yield equally elegant characterizations of log space, NC, NC^k, and other complexity classes. (Bellantoni, in [Bel92a], extends the characterization of $\mathcal{P}$ to the polynomial hierarchy. In recent personal communication he claims to have characterized log space, linear space, and NC in a similar manner, but the author has not seen the proofs.) Both Bellantoni [Bel92b] and the author [Blo92] are currently developing general logical analogues to the safe-parameters technique, which should yield proof-theoretic and model-theoretic characterizations of a variety of complexity classes in the style of Buss [Bus86].

Part II

Parameter-Free Induction

Chapter 6

Preliminaries to Part II

6.1 Scope of Part II

In Part II, which is largely independent of Part I, we study axiomatic logical systems whose proof strength corresponds to particular computational complexity classes. The fundamental intuitions in this area are that a logical sentence $(\forall\vec{x})(\exists!y)\phi(\vec{x},y)$ asserts the totality and single-valuedness of a *function* computing y from $\vec{x}$, that one can interpret a proof of the sentence as specifying an algorithm to compute the function, and that various properties of the proof correspond to the difficulty of computing the function.

A theory capable of proving the sentence $(\forall\vec{x})(\exists!y)\phi(\vec{x},y)$ is said to *define* the corresponding function. The weaker the theory is, the fewer mechanisms are available to prove the existence-and-uniqueness statement, and therefore the easier must be the computation of the function. We study what functions are provably total and unique in a given theory, and how different such theories relate to one another.

Such theories are typically subtheories of Peano arithmetic, formed by modifying the induction scheme. Many of the principal questions in this area are a matter of how much logical difference such modifications make, and how different sorts of modifications interact. The three most common modifications are to reduce the lengths of the chains of reasoning encapsulated in a single induction inference, to allow induction only on formulæ of a certain syntactic complexity, and to allow induction only with certain conditions on the occurrence of free variables.

Any induction scheme can be thought of as abbreviating a variable-length chain of successive uses of *Modus Ponens*. Given $A(0)$ and $(\forall x)(A(x) \supset A(x+1))$ and in the absence of an induction axiom we could prove $A(1)$ in one step, $A(2)$ in two, and in general $A(x)$ in x steps. Other forms of induction prove exactly the same consequences, but require stronger hypotheses that would allow us to prove $A(x)$ in $\log(x)$, or $\log(\log(x))$, or some smaller number of steps. For example, the scheme of $PIND$ (prefix induction) replaces the requirement $(\forall x)(A(x) \supset A(x+1))$ by $(\forall x)(A(\left\lfloor \frac{x}{2} \right\rfloor) \supset A(x))$; as a result the chain of reasoning abbreviated by a single $PIND$ inference is only approximately $\log(x)$ steps long.

The power of induction can also be restricted by allowing induction only on certain syntactic classes of formulæ. In a classical approach derived from recursion theory, many logicians have studied the systems with induction only on formulæ with at most a certain number of alternations between universal and existential quantifiers (typically ignoring bounded quantifiers of the form $\forall x \leq t$ or $\exists x \leq t$, as such quantifiers can be evaluated by recursive, and even primitive recursive, functions). More recently, Samuel Buss (in [Bus86]) defined analogous syntactic classes at a lower level by disallowing unbounded quantifiers, counting alternations of *bounded* quantifiers, and ignoring "sharply-bounded" quantifiers, those whose range of quantification is only logarithmic in the values of the free variables (as such quantifiers can be evaluated by deterministic polynomial-time functions).

This latter work developed two closely-related hierarchies of logical theories collectively called "theories of bounded arithmetic". The two hierarchies, $\{T_2^i\}_i$ and $\{S_2^i\}_i$, have x-step and $\log(x)$-step induction, respectively, applicable only to formulæ with at most i alternations between universal and existential bounded quantifiers. The most oft-cited theorem of [Bus86] states that a function is definable by a formula with at most i alternations and can be proven total in theory S_2^i iff it is computable at the i-th level of the polynomial-time hierarchy. In particular, the $i = 1$ case of this theorem provides an extremely useful characterization of polynomial-time-computable functions.

Various researchers (notably [All91], [CL90], [Clo89a], [CT90], [CT92], and [Fer90]) have defined similar axiomatic systems corresponding to other complexity classes. The ones that most concern us here are the systems R_2^i of [Tak92], TNC of [Clo89a] and [CT92], and D^i of [All91], which replace the $\log(x)$-step induction of S_2^i by a still weaker, $\log(\log(x))$-step induction. (Takeuti [Tak92] proves TNC and R_2^1 equivalent, and we use the latter, more flexible notation.) The systems apparently differ in that an induction in R_2^i "unrolls" into a $\log(\log(x))$-step chain, like the longer chains of S_2^i and T_2^i, while an induction inference in D^i is most naturally interpreted as a $\log(\log(x))$-depth binary tree, and therefore naturally models parallel computation. R_2^i is clearly a subtheory of D^i, for any R_2^i induction can be replaced by a D^i induction that ignores all but one leaf of its binary tree. However, the converse has also been shown (folklore, and see Section 6.6), and thus R_2^i and D^i are different axiomatizations of the same theory. As the divide-and-conquer induction of D^i is often easier to work with, though, we define the theories R_k^i for purposes of this paper in terms of it.

A third way to restrict induction is to control the occurrence of free variables in formulæ subject to induction. Richard Kaye, in [Kay87], defined *parameter-free induction* as induction in which no variables appear free except the eigenvariable, the one used directly for the induction. That is, from $A(0)$ and $(\forall x)(A(x) \supset A(x+1))$ we can conclude $(\forall x)A(x)$ as usual, but only if x is the only free variable occurring in A. (This can have interesting consequences for complexity theory. Since the usual techniques for proving a logical theory characterizes a complexity class consider computational cost as a function of *all* the free variables in a formula, parameter-free induction gives us greater control over the growth of computational costs.) Kaye proved that for several important syntactic formula classes Φ, parameter-free induction on Φ-formulæ has much the same logical consequences as induction on Φ-formulæ *with* parameters. Some of his results apply to high-level logics allowing unbounded quantifiers, while one is very similar to the bounded arithmetic theory T_2^i.

We extend his work from the x-step, Peano-style induction in [Kay87] to the weaker $\log(x)$ and $\log(\log(x))$-step induction schemes of the theories S_2^i and R_2^i. We also replace Kaye's model-theoretic approach by a proof-theoretic one; this offers three ways to improve on the earlier results. First, in the proof-theoretic setting we can make finer distinctions in "*how* parameter-free" an induction is: in Section 6.8 we distinguish between an induction axiom with no parameters, an induction inference with no parameters in principal formulæ, an induction inference with no parameters even in side formulæ, and an induction inference with no side formulæ whatsoever. Kaye's system was effectively the strongest of these four; we show that the conservation holds even with the weakest of these forms of parameter-freeness. Second, our proof-theoretic conservation preserves the important property that all formulæ appearing anywhere in the proof of a Σ_i^b endsequent are Σ_i^b, thus suggesting complexity-theoretic consequences by techniques similar to those in [Bus86]. Third, in the proof-theoretic setting we have the opportunity to address questions of proof length in formal systems, again suggesting complexity-theoretic consequences.

6.2 Data representation

Many of the results presented here are most naturally thought of in terms of bit-strings, finite strings over the alphabet $\Sigma = \{0, 1\}$. But for compatibility with related literature (notably [Bus86] and [All91]), we work formally in theories with an intended interpretation on the natural numbers. The principal difference between the two domains is that in the world of bit-strings, leading zeroes matter. We use a natural bijection between bit-strings and positive integers: a bit-string x is represented by the number whose binary form is $1x$, and inversely we can extract a bit-string from a positive integer by deleting the leading 1 bit of its binary form. (These operations will be formalized within the system by the functions $PADSTR$ and $UNPADSTR$, defined in Section 6.10.) This representation is similar to those in e.g. [Ara92] and [Clo89a].

6.3 Languages

We work in several first-order languages of arithmetic.

Definition 33 *The language $\mathcal{L}_1$ contains the following constant, function, and relation symbols:*

$$
\begin{array}{rcl}
0, 1 & & \textit{constants} \\
=, \leq & & \textit{binary infix relation symbols} \\
x + y & & \textit{ordinary addition} \\
x \dot{-} y & = & \max(0, x - y) \\
x \cdot y & & \textit{ordinary multiplication} \\
|x| & = & \lceil \log(x+1) \rceil, \ \textit{the length of } x\textit{'s binary representation} \\
\left\lfloor \dfrac{x}{\max(y,1)} \right\rfloor & & \textit{floored division, excluding zero denominators} \\
\mathit{LimExp}(x, y) & = & 2^{\min(x,|y|)}, \textit{the limited exponentiation function}
\end{array}
$$

(We seldom use the full power of division and could in fact prove all the results in this paper using only division by powers of 2, at the expense of more complicated definitions.)

This language has the property that any term in it has growth rate linear in the lengths of the free variables; that is, if $t(x)$ is a term of $\mathcal{L}_1$, then $|t(x)| = O(|x|)$. We shall also define languages $\mathcal{L}_2, \mathcal{L}_3, \ldots$ with larger growth rates by adding "smash" functions. Smash functions, originally proposed in [Hoo83] and [Nel86], and usually represented by the symbol #, are introduced into theories of bounded arithmetic purely for their growth rates — the actual value of a smash function is of little interest, though we shall assume for convenience that all the bits of the value are ones — and we define a whole family of them. Our definitions are in the style of [All91], and not quite the same as those of [Bus86] or [Hoo83]; each of the definitions can be replaced by a closed term in either of the others, so it makes little difference which we choose.

Definition 34 *The smash functions $\#_1, \#_2, \ldots$ are defined as follows:*

$$\begin{aligned} x\#_1 y &= 2^{|x|+|y|} - 1 \\ x\#_2 y &= 2^{|x|\cdot|y|} - 1 \\ x\#_3 y &= 2^{|x|\#_2|y|} - 1 \\ &\vdots \\ x\#_{i+1} y &= 2^{|x|\#_i|y|} - 1 \\ &\vdots \end{aligned}$$

Definition 35 *For $i > 1$, the language $\mathcal{L}_i$ is $\mathcal{L}_1$ augmented with the function symbols $\#_2, \#_3, \ldots \#_i$.*

Thus terms in $\mathcal{L}_1$ have linear growth rates, $\mathcal{L}_2$ polynomial growth rates, $\mathcal{L}_3$ $2^{polylog}$ growth rates, and in general terms in language $\mathcal{L}_i$ have growth rates described by a stack of $i-1$ 2's, with the final exponent being a linear combination of terms of the form $\log^{(i)}(x)$, for x a free variable, where $\log^{(i)}(x)$ denotes the $\log_2$ function iterated i times on x.

Terms and atomic formulæ of the language are constructed in the obvious manner from these symbols. They are then formed into sentences and general formulæ with the unbounded quantifiers $(\forall x)$ and $(\exists x)$ and bounded quantifiers $(\forall x \leq t(\vec{y}))$ and $(\exists x \leq t(\vec{y}))$, as well as the usual Boolean connectives $\neg, \wedge, \vee$.

We shall use the symbols $\supset$ and $\leftrightarrow$ as Boolean connectives: $A \supset B$ abbreviates $\neg A \vee B$, while $A \leftrightarrow B$ abbreviates $(A \supset B) \wedge (B \supset A)$. We can also define other useful functions and relations as abbreviations. (The reader is warned not to confuse these functions with similar ones defined in Part I.) For example,

$$\begin{aligned} x \neq y &\Leftrightarrow \neg(x = y) \\ x \geq y &\Leftrightarrow y \leq x \\ x > y &\Leftrightarrow \neg(x \leq y) \\ x < y &\Leftrightarrow \neg(y \leq x) \\ Pwr2(x) &\Leftrightarrow |x| > |x \dot{-} 1|, \text{ to test whether } x \text{ is a power of 2} \end{aligned}$$

$$
\begin{aligned}
2^{|x|} &= LimExp(|x|, x) \\
MSP(x,y) &= \left\lfloor \frac{x}{2^{|y|}} \right\rfloor, \text{ the "most significant part" of } x \\
&\quad \text{after stripping off the low } |y| \text{ bits} \\
LSP(x,y) &= x \dot{-} MSP(x,y) \cdot 2^{|y|}, \\
&\quad \text{the "least significant part", the low } |y| \text{ bits of } x \\
SUBSTR(x,y,z) &= LSP(MSP(x, LimExp(y,x) \dot{-} 1), LimExp(z,x) \dot{-} 1), \\
&\quad \text{bits } y+z, y+z-1, \ldots y \text{ of } x \\
Bit(i,x) &= SUBSTR(x,i,1), \\
&\quad \text{the bit of } x \text{ with significance } 2^i \\
Concat(x,y) &= x \cdot 2^{|y| \dot{-} 1} + y \dot{-} 2^{|y| \dot{-} 1}, \\
&\quad \text{the concatenation of the bit-strings represented by } x \text{ and } y \\
Fh(x) &= \left\lfloor \frac{x}{2^{\lfloor |x|/2 \rfloor}} \right\rfloor, \\
&\quad \text{the "front half" of the bit-string represented by } x \\
Bh(x) &= x \dot{-} Fh(x) \cdot 2^{\lfloor |x|/2 \rfloor} + 2^{\lfloor |x|/2 \rfloor}, \\
&\quad \text{the "back half" of the bit-string represented by } x
\end{aligned}
$$

The last three are defined in such a manner that if y and z are any bit-strings, $Concat(1y, 1z) = 1yz$, and if furthermore $|y| \le |z| \le |y| + 1$, then $Fh(1yz) = 1y$ and $Bh(1yz) = 1z$. (Allen, in [All91], defined slightly different Fh and Bh functions, operating on binary numbers rather than on the bit-strings represented thereby. We choose to define ours to work naturally on bit-strings because in the binary setting the "back half" of a 40-bit number can be anywhere from 0 to 20 bits long, depending on leading zeroes; in our setting it will be exactly 20 bits plus a leading 1.)

We next define syntactic classes of formulæ $\Sigma^b_{i,k}$ and $\Pi^b_{i,k}$ within $\mathcal{L}_k$. The following definitions are essentially those in [Bus86], except that since Buss was only concerned with a language analogous to $\mathcal{L}_2$, the second subscript k is our addition.

Definition 36 *A bounded quantifier is said to be* sharply bounded *if its outermost function symbol is* $|\cdots|$.

Definition 37 *The classes $\Sigma^b_{i,k}$ and $\Pi^b_{i,k}$ are the smallest sets of formulæ satisfying the following inductive definition.*

- *$\Sigma^b_{0,k}$ and $\Pi^b_{0,k}$ are both the set of formulæ in language $\mathcal{L}_k$ in which all quantifiers are sharply bounded.*
- *$\Pi^b_{i,k} \subseteq \Sigma^b_{i+1,k}$.*
- *$\Sigma^b_{i,k} \subseteq \Pi^b_{i+1,k}$.*
- *If $A \in \Sigma^b_{i+1,k}$ and t is a term in language $\mathcal{L}_k$, then the formulæ $(\exists x \leq t)A$ and $(\forall x \leq |t|)A$ are also in $\Sigma^b_{i+1,k}$.*
- *If $A \in \Pi^b_{i+1,k}$ and t is a term in language $\mathcal{L}_k$, then the formulæ $(\forall x \leq t)A$ and $(\exists x \leq |t|)A$ are also in $\Pi^b_{i+1,k}$.*
- *If $A, B \in \Sigma^b_{i,k}$ (respectively $\Pi^b_{i,k}$), then so are $A \wedge B$ and $A \vee B$.*
- *If $A \in \Sigma^b_{i,k}$ (respectively $\Pi^b_{i,k}$), then $\neg A \in \Pi^b_{i,k}$ (respectively $\Sigma^b_{i,k}$).*

When the language $\mathcal{L}_k$ is clear from context, we shall omit the subscript k.

6.4 Sequent Calculus

We shall formalize proofs within logical systems in the notation of Gentzen sequent-calculus, as in [Bus86], where a basic system called LKB was defined and then augmented with various induction axioms. We use essentially the same system LKB, except that our language contains slightly different function symbols.

Proofs in the sequent calculus are made up of *sequents*. A sequent is a comma-delimited list of zero or more formulæ, followed by a sequent arrow $\longrightarrow$ and then by another comma-delimited list of zero or more formulæ. Its intended interpretation is that for any assignment of values to the free variables appearing in the formulæ, the conjunction of the formulæ on the left materially implies the disjunction of the formulæ on the right. As usual, an empty conjunction is interpreted as "true" and an empty disjunction as "false".

Proofs in the sequent calculus are represented as trees of sequents, customarily rooted at the bottom, with the property that each non-leaf sequent is derivable by an inference rule from its children, and each leaf sequent is an instance of an axiom schema. For reference purposes we list the axiom schemas and inference rules of LKB here, but the reader is referred to [Bus86] or [Tak75] for a detailed treatment.

Logical Axioms

If A is any quantifier-free formula in the language under consideration, then

$$A \longrightarrow A$$

is a logical axiom and may appear as a leaf of a proof tree. (The "quantifier-free" stipulation is somewhat stronger than we need: in order to prove results about induction on $\Sigma^b_{i,k}$-formulæ, we need only know that all formulæ appearing in axioms are $\Sigma^b_{i,k}$. In any case, the inference rules defined below will allow us to prove any sequent of the form $A \longrightarrow A$.)

Equality Axioms

If $s_1, s_2, \ldots s_n, t_1, t_2, \ldots t_n$ are terms in the language, f is an n-ary function symbol in the language, and p is an n-ary predicate symbol in the language, then the following are equality axioms and may appear as leaves in a proof tree:

$$\begin{array}{rcl} & \longrightarrow & t_1 = t_1 \\ s_1 = t_1, \ldots s_n = t_n & \longrightarrow & f(s_1, s_2, \ldots s_n) = f(t_1, t_2, \ldots t_n) \\ s_1 = t_1, \ldots s_n = t_n, p(s_1, s_2, \ldots s_n) & \longrightarrow & p(t_1, t_2, \ldots t_n) \end{array}$$

Proper Axioms

Our theories also contain a number of proper axioms, whose main purpose is to ensure that the function symbols in the language have their intended meanings.

Buss [Bus86] gives a set of proper axioms called $BASIC$, and Allen [All91] modifies it slightly to a set $BASIC^+$, which defines the behaviour of the additional function symbols in the latter's system. Our (slightly different, yet again!) set of proper axioms follows:

(1) $\longrightarrow 0 \leq x$

(2) $\longrightarrow x \leq 1 \supset x = 0 \vee x = 1$

(3) $\longrightarrow \neg(x + 1 \leq 0)$

(4) $\longrightarrow x \leq y \vee y \leq x$

(5) $\longrightarrow (x \leq y \wedge y \leq x) \leftrightarrow x = y$

(6) $\longrightarrow (x \leq y \wedge y \leq z) \supset x \leq z$

(7) $\longrightarrow (x + y \leq x + z) \leftrightarrow y \leq z$

(8) $\longrightarrow x + 0 = x$

(9) $\longrightarrow x + (y + z) = (x + y) + z$

(10) $\longrightarrow x + y = y + x$

(11) $\longrightarrow x \leq y \supset x \dot{-} y = 0$

(12) $\longrightarrow y \leq x \supset (x \dot{-} y) + y = x$

(13) $\longrightarrow x \cdot 0 = 0$

(14) $\longrightarrow x \cdot 1 = x$

(15) $\longrightarrow x \cdot (y \cdot z) = (x \cdot y) \cdot z$

(16) $\longrightarrow x \cdot y = y \cdot x$

(17) $\longrightarrow x \cdot (y + z) = (x \cdot y) + (x \cdot z)$

(18) $\longrightarrow x \neq 0 \supset (x \cdot y \leq x \cdot z \leftrightarrow y \leq z)$

(19) $\longrightarrow |0| = 0$

(20) $\longrightarrow |x + x + 1| = |x| + 1$

(21) $\longrightarrow x \neq 0 \supset |x + x| = |x| + 1$

(22) $\longrightarrow x \leq y \supset |x| \leq |y|$

(23) $\longrightarrow Pwr2(x) \wedge y < x \supset |x + y| = |x|$

(24) $\longrightarrow Pwr2(x) \wedge Pwr2(y) \supset |x \cdot y| = |x| + |y| \dot{-} 1$

(25) $\longrightarrow Pwr2(x) \wedge Pwr2(y) \supset Pwr2(x \cdot y)$

(26) $\longrightarrow \left\lfloor \frac{x}{\max(0,1)} \right\rfloor = x$

(27) $\longrightarrow (y > 0) \supset \left(y \cdot \left\lfloor \frac{x}{\max(y,1)} \right\rfloor \leq x\right)$

(28) $\longrightarrow (y > 0) \supset \left(y \cdot \left(\left\lfloor \frac{x}{\max(y,1)} \right\rfloor + 1\right) > x\right)$

(29) $\longrightarrow Pwr2(LimExp(x, y))$

(30) $\longrightarrow |LimExp(x, y)| = \min(x, |y|) + 1$

In theories with language $\mathcal{L}_k$, for $k > 1$, we add the following:

(31) $\longrightarrow Pwr2((x \#_2 y) + 1)$

(32) $\longrightarrow |x \#_2 y| = |x| \cdot |y|$

(33) $\longrightarrow Pwr2((x \#_3 y) + 1)$

(34) $\longrightarrow |x \#_3 y| = |x| \#_2 |y|$

$\vdots$

(35) $\longrightarrow Pwr2((x\#_k y)+1)$

(36) $\longrightarrow |x\#_k y| = |x|\#_{k-1}|y|$

It is quite likely this list of axioms could be reduced, but knowing it's a fixed, finite set of quantifier-free formulæ will suffice for our purposes.

If A and B are formulæ, possibly containing free variables, and Γ, Π, Δ, and Λ are comma-delimited lists of zero or more such formulæ, then the following are valid inferences:

Structural Inferences

Weak:left

$$\frac{\Gamma \longrightarrow \Delta}{A, \Gamma \longrightarrow \Delta}$$

Weak:right

$$\frac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, A}$$

Contract:left

$$\frac{A, A, \Gamma \longrightarrow \Delta}{A, \Gamma \longrightarrow \Delta}$$

Contract:right

$$\frac{\Gamma \longrightarrow \Delta, A, A}{\Gamma \longrightarrow \Delta, A}$$

Exchange:left

$$\frac{\Gamma, A, B, \Pi \longrightarrow \Delta}{\Gamma, B, A, \Pi \longrightarrow \Delta}$$

Exchange:right

$$\frac{\Gamma \longrightarrow \Delta, A, B, \Pi}{\Gamma \longrightarrow \Delta, B, A, \Pi}$$

Propositional Inferences

¬:left

$$\frac{\Gamma \longrightarrow \Delta, A}{\neg A, \Gamma \longrightarrow \Delta}$$

¬:right

$$\frac{A, \Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \neg A}$$

∧:left

$$\frac{A, B, \Gamma \longrightarrow \Delta}{A \wedge B, \Gamma \longrightarrow \Delta}$$

∧:right

$$\frac{\Gamma \longrightarrow \Delta, A \quad \Gamma \longrightarrow \Delta, B}{\Gamma \longrightarrow \Delta, A \wedge B}$$

∨:left

$$\frac{A, \Gamma \longrightarrow \Delta \quad B, \Gamma \longrightarrow \Delta}{A \vee B, \Gamma \longrightarrow \Delta}$$

∨:right

$$\frac{\Gamma \longrightarrow \Delta, A, B}{\Gamma \longrightarrow \Delta, A \vee B}$$

⊃:left

$$\frac{\Gamma \longrightarrow \Delta, A \quad B, \Pi \longrightarrow \Lambda}{A \supset B, \Gamma, \Pi \longrightarrow \Delta, \Lambda}$$

⊃:right

$$\frac{A, \Gamma \longrightarrow \Delta, B}{\Gamma \longrightarrow \Delta, A \supset B}$$

Quantifier Inferences

$\forall$:left

$$\frac{A(t), \Gamma \longrightarrow \Delta}{(\forall x)A(x), \Gamma \longrightarrow \Delta}$$

$\forall$:right

$$\frac{\Gamma \longrightarrow \Delta, A(a)}{\Gamma \longrightarrow \Delta, (\forall x)A(x)}$$

where a is a free variable that appears only as specified.

$\exists$:left

$$\frac{A(a), \Gamma \longrightarrow \Delta}{(\exists x)A(x), \Gamma \longrightarrow \Delta}$$

where again a is a free variable that appears only as specified.

$\exists$:right

$$\frac{\Gamma \longrightarrow \Delta, A(t)}{\Gamma \longrightarrow \Delta, (\exists x)A(x)}$$

$\forall \leq$:left

$$\frac{A(t), \Gamma \longrightarrow \Delta}{t \leq s, (\forall x \leq s)A(x), \Gamma \longrightarrow \Delta}$$

$\forall \leq$:right

$$\frac{a \leq t, \Gamma \longrightarrow \Delta, A(a)}{\Gamma \longrightarrow \Delta, (\forall x \leq t)A(x)}$$

where a is a free variable that appears only as specified.

$\exists \leq$:left

$$\frac{a \leq t, A(a), \Gamma \longrightarrow \Delta}{(\exists x \leq t)A(x), \Gamma \longrightarrow \Delta}$$

where again a is a free variable that appears only as specified.

$\exists \leq$:right

$$\frac{\Gamma \longrightarrow \Delta, A(t)}{t \leq s, \Gamma \longrightarrow \Delta, (\exists x \leq s)A(x)}$$

The Cut Inference

$$\frac{\Gamma \longrightarrow \Delta, A \quad A, \Pi \longrightarrow \Lambda}{\Gamma, \Pi \longrightarrow \Delta, \Lambda}$$

This "cut" inference is the analogue to the rule of *Modus Ponens* in a Hilbert-style deduction system. Its principal importance in constructive mathematics is that, alone among all the inference rules of LKB, it allows formulæ above the line that do not appear at all below the line. More precisely, all the inference rules except "cut" have the property that every formula in the hypotheses is a substitution instance of a subformula of the conclusion. Thus in a system without cuts, one could search for proofs of a given formula simply by enumerating its subformulæ and testing which inference rules could possibly apply. Although term substitution could still make the subformulæ appearing in a proof arbitrarily long, their *logical* complexity would be bounded by that of the formula to be proven.

As it turns out, a principal theorem of Gentzen's work on sequent calculus [Gen35] states that in fact we can assume without loss of generality that there are only certain restricted kinds of cuts in a proof. This statement will be made more precise in Theorem 26 below.

6.5 Types of Induction

We shall seldom actually use *LKB* itself, but rather *theories of bounded arithmetic* defined from it by adding various sorts of induction, either as inference rules or as proper axioms. Each is normally defined to apply to only a certain syntactic class of formulæ Φ, typically $\Sigma^b_{i,k}$ for some i, k. In presenting these inference rules and axioms we shall explicitly list *all* occurrences of free variables so as to clearly distinguish between their parameter-free and non-parameter-free forms. In all the induction rules and axioms it is also assumed that $A \in \Phi$.

$\Phi - IND$ rule:

$$\frac{\Gamma(\vec{b}), A(a, \vec{b}) \quad \longrightarrow \quad A(a+1, \vec{b}), \Delta(\vec{b})}{\Gamma(\vec{b}), A(0, \vec{b}) \quad \longrightarrow \quad A(t(\vec{b}), \vec{b}), \Delta(\vec{b})}$$

$\Phi - PIND$ rule:

$$\frac{\Gamma(\vec{b}), A(\lfloor \frac{a}{2} \rfloor, \vec{b}) \quad \longrightarrow \quad A(a, \vec{b}), \Delta(\vec{b})}{\Gamma(\vec{b}), A(0, \vec{b}) \quad \longrightarrow \quad A(t(\vec{b}), \vec{b}), \Delta(\vec{b})}$$

$\Phi - LIND$ rule:

$$\frac{\Gamma(\vec{b}), A(a, \vec{b}) \quad \longrightarrow \quad A(a+1, \vec{b}), \Delta(\vec{b})}{\Gamma(\vec{b}), A(0, \vec{b}) \quad \longrightarrow \quad A(|t(\vec{b})|, \vec{b}), \Delta(\vec{b})}$$

$\Phi - DCI$ rule:

$$\frac{\Gamma(\vec{b}), A(Bh(a), \vec{b}), A(Fh(a), \vec{b}) \quad \longrightarrow \quad A(a, \vec{b}), \Delta(\vec{b})}{\Gamma(\vec{b}), A(0, \vec{b}), A(1, \vec{b}), A(2, \vec{b}), A(3, \vec{b}) \quad \longrightarrow \quad A(t(\vec{b}), \vec{b}), \Delta(\vec{b})}$$

$\Phi - PPIND$ rule:

$$\frac{\Gamma(\vec{b}), A(Fh(a), \vec{b}) \quad \longrightarrow \quad A(a, \vec{b}), \Delta(\vec{b})}{\Gamma(\vec{b}), A(0, \vec{b}), A(1, \vec{b}) \quad \longrightarrow \quad A(t(\vec{b}), \vec{b}), \Delta(\vec{b})}$$

$\Phi - LPIND$ rule:

$$\frac{\Gamma(\vec{b}), A(\lfloor \frac{a}{2} \rfloor, \vec{b}) \quad \longrightarrow \quad A(a, \vec{b}), \Delta(\vec{b})}{\Gamma(\vec{b}), A(0, \vec{b}) \quad \longrightarrow \quad A(|t(\vec{b})|, \vec{b}), \Delta(\vec{b})}$$

$\Phi - LLIND$ rule:

$$\frac{\Gamma(\vec{b}), A(a, \vec{b}) \quad \longrightarrow \quad A(a+1, \vec{b}), \Delta(\vec{b})}{\Gamma(\vec{b}), A(0, \vec{b}) \quad \longrightarrow \quad A(||t(\vec{b})||, \vec{b}), \Delta(\vec{b})}$$

Of these forms of induction the first, IND, is the familiar form of mathematical induction. $LIND$ and $LLIND$ are clearly special cases of it, only allowed to induct up to values in the range of the length (respectively, length of length) function. $PIND$, introduced in this form in [Bus86], abbreviates a sequence of steps each of which asserts that A holds of a if it holds of the prefix of a formed by "chopping off the low bit"; the resulting sequence therefore contains only $|t(\vec{b})|$ steps. $PPIND$ is similar, but rather than regressing on one-bit-shorter prefixes, it regresses on a prefix whose length is approximately half that of a; the resulting sequence therefore contains only $\log_2(|t(\vec{b}|))$ steps. It includes a base case of 1 as well as 0 to prevent infinite regress, since under our definitions $Fh(1) = 1$. The DCI ("divide and conquer induction") rule, introduced in [All91], regresses *simultaneously* on both the front half and the back half of a, and is familiar to computer scientists as the natural way of proving correctness of a divide-and-conquer algorithm like Mergesort. In our formulation, which is slightly different from Allen's, we must leave the values 0,1,2, and 3 as base cases, again to prevent infinite regress since for example $Bh(3) = 3$. Finally, $LPIND$ has the hypotheses of PIND and the conclusion of $LIND$.

6.6 Theories of Bounded Arithmetic

Buss [Bus86] defines the logical theories $\{T^i_k\}_{i,k}$ and $\{S^i_k\}_{i,k}$ by augmenting LKB with the $\Sigma^b_{i,k}$-IND and $\Sigma^b_{i,k}$-PIND inference rules, respectively. The theories $\{D^i\}_i$ of [All91] are defined similarly by augmenting LKB with the $\Sigma^b_{i,2}$-DCI inference rule. Takeuti [Tak92], defines theories $\{R^i_k\}_{i,k}$ by augmenting LKB with the $\Sigma^b_{i,k}$-LPIND inference rule. Clote [Clo89a] defines a theory TNC by augmenting LKB with the $\Sigma^b_{1,2}$-LLIND inference rule, as well as a $\Pi^b_{1,2}$-separation rule and an axiom of extensionality; Allen [All91] and Clote-Takeuti [CT92] independently prove the separation rule superfluous, and we prove the extensionality axiom superfluous. Buss, Krajíček, and Takeuti [BKT92] use the LPIND definition.

A number of relationships are known among these theories. First, since Φ-

LLINDis a special case of Φ-LIND which is in turn a special case of Φ-IND, clearly Φ-IND $\vdash$ Φ-LIND $\vdash$ Φ-LLIND for any syntactic class Φ, and similarly Φ-PIND $\vdash$ Φ-LPIND.

Less trivially, it can be proven ([Bus86], [Tak92]) that in the presence of a few common function symbols and their defining axioms (see Theorems 24 and 25 for details) Φ-LIND and Φ-PIND are equivalent, and Φ-LLIND, Φ-LPIND, Φ-PPIND, and Φ-DCI are all equivalent, and thus that

$$T_k^i \supseteq S_k^i \supseteq R_k^i$$

for $i > 0$. The converses of these inclusions are open, and seem unlikely: for example, $T_2^i \subseteq S_2^i$ is false when relativized with a suitable predicate [BK92]. It is known that $S_k^{i+1} \supseteq T_k^i$ [Bus86] and $R_k^{i+1} \supseteq S_k^i$ (Takeuti), again for $i > 0$. The converses of these inclusions, too, are open and seem unlikely: $S_2^{i+1} \subseteq T_2^i$, for example, would imply the collapse of the polynomial-time hierarchy to level $i + 2$ [KPT91].

Theorem 24 (Buss) *If a theory T in language $\mathcal{L}_k$ contains the proper axioms of Section 6.4, then for each $i > 0$, $T \vdash \Sigma_{i,k}^b$-LIND if and only if $T \vdash \Sigma_{i,k}^b$-PIND.*

Proof that $\Sigma_{i,k}^b$-PIND proves $\Sigma_{i,k}^b$-LIND: Suppose $A(x, \vec{y}) \in \Sigma_{i,k}^b$, and let $B(x, \vec{y}) \Leftrightarrow A(|x|, \vec{y})$. Then clearly $A(0, \vec{y}) \Rightarrow B(0, \vec{y})$ and $B(t, \vec{y}) \Rightarrow A(|t|, \vec{y})$. Furthermore, given the induction hypothesis $(\forall x)(A(x, \vec{y}) \supset A(x + 1, \vec{y}))$, we have

$$\begin{aligned} B\left(\left\lfloor \frac{a}{2} \right\rfloor, \vec{y}\right) &\Rightarrow A\left(\left|\left\lfloor \frac{a}{2} \right\rfloor\right|, \vec{y}\right) \\ &\Rightarrow A(|a| \dot{-} 1, \vec{y}) \\ &\Rightarrow A(|a|, \vec{y}) \\ &\Rightarrow B(a, \vec{y}) \end{aligned}$$

so the induction hypothesis is satisfied for PIND on B. Thus we conclude $B(0, \vec{y}) \Rightarrow B(t, \vec{y})$ and hence $A(0, \vec{y}) \Rightarrow A(|t|, \vec{y})$ as desired.

Proof that $\Sigma_{i,k}^b$-LIND proves $\Sigma_{i,k}^b$-PIND: Suppose $A(x, \vec{y}) \in \Sigma_{i,k}^b$, and let $B(x, u, \vec{y}) \Leftrightarrow A\left(\lfloor \frac{x}{2^{|x| \dot{-} u}} \rfloor, \vec{y}\right)$. Then clearly $A(0, \vec{y}) \Rightarrow B(t, 0, \vec{y})$ and $B(t, |t|, \vec{y}) \Rightarrow A(t, \vec{y})$. Furthermore, given the induction hypothesis $(\forall x)(A(\lfloor \frac{x}{2} \rfloor, \vec{y}) \supset A(x, \vec{y}))$, we have

$$\begin{aligned} B(t,a,\vec{y}) &\Rightarrow A\left(\left\lfloor \frac{t}{2^{|t| \dot{-} a}} \right\rfloor, \vec{y}\right) \\ &\Rightarrow A\left(\left\lfloor \frac{t}{2^{|t| \dot{-} (a+1)}} \right\rfloor, \vec{y}\right) \\ &\Rightarrow B(t, a+1, \vec{y}) \end{aligned}$$

and so the induction hypothesis is satisfied for IND or LIND on B. By LIND we conclude $B(t, 0, \vec{y}) \Rightarrow B(t, |t|, \vec{y})$ and hence $A(0, \vec{y}) \Rightarrow A(t, \vec{y})$ as desired. ∎

Note that PIND proves LIND without introducing new variables, but the converse does not appear to be the case. Thus "parameter-free" PIND (without the side variables $\vec{y}$) proves a similarly parameter-free LIND, but not obviously *vice versa*.

Theorem 25 (Folklore (Takeuti?)) *If a theory T in language $\mathcal{L}_k$ contains the proper axioms of Section 6.4, then for each $i > 0$, the following are equivalent:*

- $T \vdash \Sigma^b_{i,k}$*-LLIND*
- $T \vdash \Sigma^b_{i,k}$*-LPIND*
- $T \vdash \Sigma^b_{i,k}$*-PPIND*
- $T \vdash \Sigma^b_{i,k}$*-DCI.*

Proof that $\Sigma^b_{i,k}$-LPIND proves $\Sigma^b_{i,k}$-LLIND: Given $A(x, \vec{y}) \in \Sigma^b_{i,k}$ on which we wish to perform LLIND, let $B(x, \vec{y}) \Leftrightarrow A(|x|, \vec{y})$. The rest of the proof is straightforward, following the model of Theorem 24.

Proof that $\Sigma^b_{i,k}$-PPIND proves $\Sigma^b_{i,k}$-LPIND: Suppose $A(x, \vec{y}) \in \Sigma^b_{i,k}$, and define the formula $B(x, \vec{y})$ to be $A(|x| \dot{-} 1, \vec{y})$. Then clearly $A(0, \vec{y}) \Rightarrow B(0, \vec{y}) \wedge B(1, \vec{y})$ and $B(2t, \vec{y}) \Rightarrow A(|t|, \vec{y})$. The rest of the proof is straightforward, following the above model.

Proof that $\Sigma^b_{i,k}$-DCI proves $\Sigma^b_{i,k}$-PPIND: This doesn't even require changing the formula; the conclusions of DCI and PPIND are the same, and the hypotheses of PPIND are stronger.

Proof that $\Sigma^b_{i,k}$-LLIND proves $\Sigma^b_{i,k}$-DCI: Suppose $A(x,\vec{y}) \in \Sigma^b_{i,k}$, and define the terms

$$S(x,u,i) = \left\lceil \frac{(|x| \dot{-} 1)i}{LimExp(u,x)} \right\rceil,$$
$$L(x,u,i) = S(x,u,i+1) - S(x,u,i),$$

and

$$D(x) = \lceil \log_2(|x| \dot{-} 1) \rceil$$

for $|x| > 2$. $S(x,u,i)$ and $L(x,u,i)$ are intended to represent the starting point and the length, respectively, of the $i+1$-st block in the division of x into 2^u blocks by the Fh and Bh functions, while $D(x)$ represents the maximum depth of such a division, as can be seen by plugging in $D(x)$ for u:

$$\begin{aligned} S(x,D(x),i) &= \left\lceil \frac{(|x| \dot{-} 1)i}{2^{\lceil \log(|x| \dot{-} 1) \rceil}} \right\rceil \\ &\leq \left\lceil \frac{(|x| \dot{-} 1)i}{|x| \dot{-} 1} \right\rceil \\ &\leq i \end{aligned}$$

Furthermore, since

$$\frac{|x| \dot{-} 1}{2^{\lceil \log(|x| \dot{-} 1) \rceil}} \leq 1,$$

it follows that $L(x,D(x),i) = S(x,D(x),i+1) - S(x,D(x),i) \leq 1$.

However, we shall want to count levels from the bottom, rather than the top, so we define $H(x,w) = D(x) \dot{-} w$. Thus $L(x,H(x,0),i) \leq 1$ for all i. Clearly $S(x,u,i)$ and $L(x,u,i)$ are bounded by $|x|$, so we can exponentiate them legally; likewise, we can double-exponentiate $H(x,u)$ and still remain bounded by x, or exponentiate $H(x,u)$ and still have something sharply bounded. Then let $BLOCK(x,w,i)$ be the term

$$SUBSTR(x, S(x,H(x,w),i), L(x,H(x,w),i)) + 2^{L(x,H(x,w),i)}$$

and let $B(x,w,\vec{y})$ be the formula

$$\left(\forall i < 2^{H(x,w)}\right) A(BLOCK(x,w,i),\vec{y}).$$

This states, intuitively, that A holds of all the blocks w levels up from the bottom. The blocks for $w = 0$, in particular, are guaranteed to be at most length 1 (length 2 counting the leading 1 bit), so

$$A(0, \vec{y}) \wedge A(1, \vec{y}) \wedge A(2, \vec{y}) \wedge A(3, \vec{y}) \Rightarrow B(x, 0, \vec{y}).$$

Similarly, when $u = 0$ (that is, when $w = D(x) + 1$), the block length is $|x| \dot{-} 1$, and so all of x is a single block, so $B(t, D(t) + 1, \vec{y}) \Rightarrow A(t, \vec{y})$. Now, given the induction hypothesis $(\forall x)(A(Fh(x), \vec{y}) \wedge A(Bh(x), \vec{y}) \supset A(x, \vec{y}))$, we have

$$\begin{aligned}
B(x, w, \vec{y}) &\Rightarrow \left(\forall i < 2^{H(x,w)}\right)(A(BLOCK(x, w, i), \vec{y}) \\
&\Rightarrow \left(\forall i < 2^{H(x,w+1)}\right) \\
&\qquad (A(BLOCK(x, w, 2i), \vec{y}) \wedge A(BLOCK(x, w, 2i + 1), \vec{y})) \\
&\Rightarrow \left(\forall i < 2^{H(x,w+1)}\right) \\
&\qquad (A(Fh(BLOCK(x, w + 1, i))) \wedge A(Bh(BLOCK(x, w + 1, i)))) \\
&\Rightarrow \left(\forall i < 2^{H(x,w+1)}\right)(A(BLOCK(x, w + 1, i))) \\
&\Rightarrow B(x, w + 1, \vec{y})
\end{aligned}$$

so the induction hypothesis is satisfied for LLIND on B. Since $D(t) + 1$ is double-sharply-bounded, we can induct up to it and conclude $B(t, 0, \vec{y}) \Rightarrow B(t, D(t) + 1, \vec{y})$ and hence $A(0, \vec{y}) \Rightarrow A(t, \vec{y})$ as desired. ∎

Note again that without introducing new variables DCI proves PPIND, which proves LPIND, which proves LLIND; none of the reverse implications is apparent.

6.7 Free Cuts

We define the *side formula occurrences* in an inference as the formula occurrences in the lists Γ, Π, Δ, and Λ, the ones which are not directly affected by the inference, while the *principal formula occurrences* are the rest, the ones distinguished as A, B, $A \vee B$, etc. in the definition of the inference rule. Following Buss [Bus86] and [Tak75], we define the *successor* (in a given proof) of an occurrence of a formula as follows:

- A side formula occurrence in a top sequent of any inference rule has as successor the corresponding formula occurrence in the bottom sequent of the rule.
- Each of the principal formula occurrences in the top sequent of a contraction rule has as successor the principal formula occurrence in the bottom sequent.
- The principal formulæ A and B in the top sequent of an exchange rule have the formulæ A and B respectively in the bottom sequent as their successors.
- Each of the principal formulæ in the top sequent(s) of a propositional inference has as successor the principal formula in the bottom sequent.
- The principal formula $A(t)$ or $A(a)$ in the top sequent of a quantifier inference has as successor the quantified principal formula in the bottom sequent.
- The principal formula $a \leq t$ in the top sequent of a $\forall \leq$:right or $\exists \leq$:left inference has no successor.

Formula occurrence A is a *predecessor* of formula occurrence B if B is a successor of A; if furthermore A and B are identical formulæ, A is said to be a *direct predecessor* of B. (In general this happens only when A and B are side formulæ or the inference is structural.) The transitive closures of these relations are called *ancestor* and *direct ancestor* relations respectively. A cut inference is defined as *constrained* if at least one of its principal formulæ has a direct ancestor which is either a formula in an axiom or a principal formula of an induction inference. Otherwise the cut inference is said to be *free*. A constrained cut, intuitively, depends on properties of a specific system such as its proper axioms and its induction capability. On the other hand, in a free cut both principal formulæ have been produced from simpler ones by purely first-order-logic operations. The next theorem shows that any free cut can be replaced by cuts on these simpler formulæ, and furthermore shows by iterating the process that any proof can be converted into a form without free cuts.

Theorem 26 (Buss, after Gentzen and Takeuti) *If P is a proof of the sequent $\Gamma \longrightarrow \Delta$ in a theory of bounded arithmetic T, then there is a T-proof P' of $\Gamma \longrightarrow \Delta$ containing no free cuts.*

This theorem assists in the study of the constructive content of bounded-arithmetic proofs, because if we restrict induction to apply only to $\Sigma^b_{i,k}$-formulæ, and if all our axioms are $\Sigma^b_{i,k}$, then any free-cut-free proof of a $\Sigma^b_{i,k}$ conclusion contains only formulæ in $\Sigma^b_{i,k} \cup \Pi^b_{i,k}$.

We omit the proof of this theorem (which may be found in e.g. [Tak75]), but point out that it provides a recursive construction of P' from P. Unfortunately, the correctness of this recursive construction is proven by transfinite induction (up to ordinal ω^2), and in general the proof P' is superexponentially larger than P (see [Sta78],[Pud85]). Since the theories we work with cannot prove the totality of a function with even exponential growth (by Parikh's theorem; see section 6.9), this theorem is not provable in our theories, and hence not practically constructive.

6.8 Parameter-Freeness

Now we define a variety of parameter-free induction rules, analogous to the induction rules defined in Section 6.5, and corresponding parameter-free theories of bounded arithmetic. The *weakly parameter-free* analogue of an induction inference requires the principal formulæ in the upper sequent to contain no free variables other than the eigenvariable, the variable on which the induction operates. The *moderately parameter-free* analogue requires the upper sequent to contain no free variables at *all* other than the eigenvariable, even in side formulæ. Finally, the *strongly parameter-free* analogue requires the upper sequent to contain neither free variables nor side formulæ. We illustrate these for IND; the remaining definitions are similar. As in Section 6.5, free variables occur only as indicated.

weakly parameter-free Φ-IND rule (Φ-wpfIND):

$$\frac{\Gamma(\vec{b}), A(a) \longrightarrow A(a+1), \Delta(\vec{b})}{\Gamma(\vec{b}), A(0) \longrightarrow A(t(\vec{b})), \Delta(\vec{b})}$$

moderately parameter-free Φ-IND rule (Φ-mpfIND):

$$\frac{\Gamma, A(a) \longrightarrow A(a+1), \Delta}{\Gamma, A(0) \longrightarrow A(t(\vec{b})), \Delta}$$

strongly parameter-free Φ-IND rule (Φ-spfIND):

$$\frac{A(a) \longrightarrow A(a+1)}{A(0) \longrightarrow A(t(\vec{b}))}$$

Parameter-free induction could also be presented as an axiom scheme, as in [Kay87]. Again, we show the IND version; the rest are similar.

parameter-free Φ-IND (Φ-pfIND) axiom:

$$A(0), (\forall x)(A(x) \supset A(x+1)) \longrightarrow A(t(\vec{b}))$$

If we are to talk about parameter-free theories of bounded arithmetic, then, the first question is which of these four definitions (the weakly parameter-free rule, the moderately parameter-free rule, the strongly parameter-free rule, or the parameter-free axiom) will be most fruitful. We approach this by comparing the logical strengths of the four definitions.

The proof of Theorem 26 breaks down when we require induction to be parameter-free (by any definition), as transforming a proof into a cut-free version involves pushing logical inferences up through inductions, and it is not clear that this can be done without introducing a free variable into the induction. So we are concerned with whether proofs using parameter-free induction are free-cut-free.

It is obvious that Φ-wpfIND can simulate Φ-mpfIND, which can in turn simulate Φ-spfIND without introducing new free cuts (and similarly for PIND, DCI, etc.), because each is a special case of the previous one. We can partially reverse these simulations.

Theorem 27 *For Φ a syntactic class of formulæ, any proof P using Φ-wpfIND can be converted to a proof P' using the Φ-pfIND axiom. If P was free-cut-free, P' will be free-cut-free as well.*

Any proof P using the Φ-pfIND axiom can be converted to a proof P' using Φ-spfIND, but not necessarily preserving free-cut-freeness.

The same holds with IND replaced throughout by PIND, DCI, etc.

Proof that the $\Phi - pfIND$ axiom can simulate the $\Phi - wpfIND$ inference, preserving free-cut-freeness:

$$\dfrac{\dfrac{\dfrac{\Gamma(\vec{b}), A(a) \longrightarrow A(a+1), \Delta(\vec{b})}{\Gamma(\vec{b}) \longrightarrow A(a) \supset A(a+1), \Delta(\vec{b})}}{\Gamma(\vec{b}) \longrightarrow (\forall x)(A(x) \supset A(x+1)), \Delta(\vec{b})} \quad A(0), (\forall x)(A(x) \supset A(x+1)) \longrightarrow A(t(\vec{b}))}{\Gamma(\vec{b}), A(0) \longrightarrow A(t(\vec{b})), \Delta(\vec{b})}$$

Note that the the cut inference here is constrained, and that the formulæ $A(0)$ and $A(t(\vec{b}))$ in the endsequent are directly descended from the axiom and hence can be cut with impunity just as though they had come from a $\Phi - wpfIND$ inference. Thus no new free cuts are introduced by replacing a $\Phi - wpfIND$ inference by the above block of proof.

Proof that $\Phi - spfIND$ proves the $\Phi - pfIND$ axiom:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A(a) \longrightarrow A(a)}{\neg A(a), A(a) \longrightarrow}}{\neg A(a), A(a) \longrightarrow A(a+1)} \quad \dfrac{A(a+1) \longrightarrow A(a+1)}{A(a+1), A(a) \longrightarrow A(a+1)}}{\neg A(a) \vee A(a+1), A(a) \longrightarrow A(a+1)}}{A(a) \supset A(a+1), A(a) \longrightarrow A(a+1)}}{(\forall x)(A(x) \supset A(x+1)), A(a) \longrightarrow A(a+1)}}{(\forall x)(A(x) \supset A(x+1)), A(0) \longrightarrow A(a)}$$

This time, although the above block of proof itself contains no free cuts, replacing a $\Phi - pfIND$ axiom by it may not preserve free-cut-freeness because the formula

$(\forall x)(A(x) \supset A(x+1))$ in the conclusion is not directly descended from an induction or axiom, and so cannot necessarily be cut constrainedly as it could if it appeared in an axiom.

The proofs of both statements for $PIND$, DCI, etc. are similar. ∎

Thus the axiom and the three rules are equivalent in their ability to prove any given theorem, but if $\underset{FCF}{\vdash}$ denotes *free-cut-free* provability, we have for any formula θ

$$(spfIND \underset{FCF}{\vdash} \theta) \Rightarrow (mpfIND \underset{FCF}{\vdash} \theta) \Rightarrow (wpfIND \underset{FCF}{\vdash} \theta) \Rightarrow (pfIND \underset{FCF}{\vdash} \theta)$$

and it is not obvious that any of the implications can be reversed.

The above theorem is unsatisfying from a complexity-theoretic viewpoint. Buss [Bus86] makes proofs free-cut-free in order to ensure that all formulæ in the proof are in Φ, yet the first proof above seems to depend on cutting a $\forall(\Phi \supset \Phi)$ formula, while the second seems to depend just as essentially on a $\forall(\Phi \supset \Phi)$ side formula passing through the induction inference.

The main theorems of Part II address both of these problems, showing (for appropriate Φ) that if we require not free-cut-freeness but only the weaker property that all formulæ in a proof are in $\Phi \cup \neg\Phi$, and if we restrict our attention to $\theta \in \Phi$, not only can the implications be reversed but all four of these parameter-free induction schemes can simulate *non*-parameter-free Φ-induction.

We accordingly define the parameter-free theories of bounded arithmetic to be as weak as possible. Let pfT^i_k be the theory in language $\mathcal{L}_k$ with induction inference rule $\Sigma^b_{i,k} - spfIND$, pfS^i_k the theory in language $\mathcal{L}_k$ with induction inference rule $\Sigma^b_{i,k} - spfPIND$, and pfR^i_k the theory in language $\mathcal{L}_k$ with induction inference rule $\Sigma^b_{i,k} - spfDCI$.

6.9 Defining New Functions and Relations

One important fact about theories of bounded arithmetic (both parameter-free and otherwise) is an instance of Parikh's theorem:

Theorem 28 (Parikh) *If T is a theory of bounded arithmetic, A a formula all of whose quantifiers are bounded, and*

$$T \vdash (\forall \vec{x})(\exists y)A(\vec{x}, y),$$

then there is a term t in the language of T such that

$$T \vdash (\forall \vec{x})(\exists y \leq t(\vec{x}))A(\vec{x}, y)$$

In other words, if T proves the totality of a function, it actually proves a closed-form bound on the value of the function.

We often wish to define new functions and relations and add them to the language. If the new function or relation in question can be defined as an abbreviation built from the existing symbols, as in the cases of $Pwr2$,MSP,LSP,Fh,Bh,$\neq$, and so on, this is easy. But many functions not so easily definable can nevertheless be added to the language without changing its expressive power.

We say a formula $A(\vec{x}, y)$ defines a function f on the natural numbers if for all values of $\vec{n}, m \in \mathbf{N}$, $A(\vec{n}, m)$ holds iff $f(\vec{n}) = m$. Similarly, a formula $A(\vec{x})$ is said to define a relation R on the natural numbers if for all values of $\vec{n} \in \mathbf{N}$, $A(\vec{n})$ holds iff $R(\vec{n})$ does. A theorem from [Bus86] allows us to use such definitions to effectively extend our language.

Definition 38 *If T is a theory of bounded arithmetic whose induction scheme applies to all and only Σ_i^b-formulæ for some $i > 0$, and if formula A defines a function f, then T_f denotes the theory whose language is that of T augmented by the function symbol f, whose axioms are those of T augmented by the axiom $f(\vec{x}) = y \leftrightarrow A(\vec{x}, y)$, and whose induction applies to all and only $\Sigma_i^b(f)$ formulæ, i.e. Σ_i^b-formulæ in the expanded language.*

Similarly, if A defines a relation R, then T_R denotes the theory whose language is that of T augmented by the relation symbol R, whose axioms are those of T augmented by the axiom $R(\vec{x}) \leftrightarrow A(\vec{x})$, and whose induction applies to all and only $\Sigma_i^b(R)$ formulæ.

Definition 39 (Buss) *If T is a theory of bounded arithmetic, $A \in \Sigma_1^b$ defines a function f, and $T \vdash (\forall \vec{x})(\exists! y)A(\vec{x}, y)$, then we say T Σ_1^b-defines f.*

If T is a theory of bounded arithmetic, $A \in \Sigma_1^b$, $B \in \Pi_1^b$, both A and B define a relation R, and $T \vdash (\forall \vec{x})(A(\vec{x}) \leftrightarrow B(\vec{x}))$, then we say T Δ_1^b-defines R.

Theorem 29 (Buss) *If T Σ_1^b-defines a function f, then the theory T_f is a conservative extension of T. In particular, anything provable by induction on a formula containing f is in fact provable in T.*

If T Δ_1^b-defines a relation R, then the theory T_R is a conservative extension of T. In particular, anything provable by induction on a formula containing R is in fact provable in T.

We omit the formal proof of this theorem (which applies equally well to parameter-free theories), but illustrate it. The fundamental intuition is that for $i > 0$, any Σ_i^b-formula in the language of T_f or T_R can be replaced by one in the language of T, provably equivalent to the first in T_f or T_R respectively. Since the induction rules applicable in T depend only on the quantifier complexity of the induction formula, any valid induction in T_f or T_R remains valid when this transformation is made.

Suppose a proof in theory T_f involves induction on the formula F in the language of T_f. The inference rules and proper axioms of LKB are sufficient to prove F equivalent to a prenex form of F, so we may as well assume F is in prenex form; in particular, since $i > 0$, every atomic subformula of F occurs inside the scope of at least one quantifier. Working from the inside out, we replace occurrences of f as follows. Recall that since $T \vdash (\forall \vec{x})(\exists! y)A(\vec{x}, y)$, Parikh's theorem tells us there is a term s such that

$$T \vdash (\forall \vec{x})(\exists! y \leq s(\vec{x}))A(\vec{x}, y)$$

Now if $f(t_1, \ldots t_n)$ appears in an atomic formula $C(f(\vec{t}))$ occurring positively in the scope of a bounded existential quantifier (or negatively in the scope of a bounded universal quantifier), then we replace $C(f(\vec{t}))$ by

$$(\exists y \leq s(\vec{t}))(A(\vec{t}, y) \wedge C(y))$$

Similarly, if $C(f(\vec{t}))$ appears positively in the scope of a bounded universal quantifier (or negatively in the scope of a bounded existential quantifier), we replace $C(f(\vec{t}))$ by

$$(\forall y \leq s(\vec{t}))(A(\vec{t}, y) \supset C(y))$$

Both of these replacements are provably equivalent to $C(f(\vec{t}))$ in T_f because T (and therefore T_f) proves the bounded existence and uniqueness of the y such that $A(\vec{x}, y)$. And neither increases the quantifier alternation of the formula F, because the Σ_1^b-formula A appears either positively inside bounded existentials, or negatively inside bounded universals.

The other half of the theorem is similar. Since the relation R is provably equivalent to both a Σ_1^b and a Π_1^b formula, we can replace it by either one, depending on whether the R appears positively or negatively in the scope of an existential or universal quantifier. Thus, like the function symbol f, the relation symbol R can be eliminated without increasing the quantifier depth of formulæ containing it (so long as they're at least $\Sigma_1^b \cup \Pi_1^b$).

6.10 Bootstrapping

With this mechanism for extending the language, we now set about defining a variety of useful functions and relations, and proving useful properties about them. Many of the following results were proven for S_2^1 (albeit with a different language) in [Bus86], but we want them for weaker theories such as R_1^1, R_2^1, and S_1^1.

The relation $Pwr2(x)$ was defined above in Section 6.3. One can prove such properties as $Pwr2(x) \supset Pwr2(x + x)$ and $Pwr2(x) \wedge Pwr2(y) \wedge |x| = |y| \supset x = y$ without induction.

We can likewise prove, without induction, that the $Concat$ function defined in Section 6.3 is associative, that 1 (which represents the empty bit string) is a two-sided identity for it, and that $Concat(Fh(x), Bh(x)) = x$.

The relation $Divides(x, y)$, indicating that x divides y, can likewise be defined as an abbreviation of $\left\lfloor \frac{y}{\max(x,1)} \right\rfloor \cdot x = y$. We can easily prove such properties as

$$Divides(x,y) \wedge Divides(x,y') \supset Divides(x,y+y')$$

and

$$Divides(x,y) \supset Divides(x,y \cdot z)$$

without induction.

The functions $\max(x,y)$ and $\min(x,y)$ can be defined in a quantifier-free (hence certainly Σ_1^b) way:

$$\begin{aligned} \max(x,y) = z &\leftrightarrow (x > y \wedge x = z) \vee (x \leq y \wedge y = z) \\ \min(x,y) = z &\leftrightarrow (x > y \wedge y = z) \vee (x \leq y \wedge x = z) \end{aligned}$$

It is easily proven (without induction) that both these functions are total and single-valued.

With max thus defined, we define $Diff(x,y) = \max(x \dot{-} y, y \dot{-} x)$; clearly $x \neq y \leftrightarrow Diff(x,y) > 0$. We then define a function $BitDiff(x,y)$ which, given two distinct numbers, finds a bit where they differ. (If $x = y$, it returns 0.) Its logical definition is

$$\begin{aligned} &BitDiff(x,y) = z \leftrightarrow \\ &\quad \leftrightarrow \ (x = y \wedge z = 0) \vee \\ &\qquad Pwr2(z) \wedge Divides(z, Diff(x,y)) \wedge \neg Divides(2z, Diff(x,y))). \end{aligned}$$

This function is clearly single-valued, since $BitDiff(x,y) = z$ implies that no multiple of $2z$ satisfies the requirements, and every power of 2 greater than z is a multiple of $2z$. We prove its totality in pfR_1^1 by DCI on d in the formula $\phi(d)$ defined as

$$((\exists z \leq d)(Pwr2(z) \wedge Divides(z,d) \wedge \neg Divides(2z,d)) \vee (d = 0).$$

The "$\vee(d = 0)$" takes care of the base case $d = 0$, and the base cases $d = 1, 2, 3$ are easy (let $z = 1, 2, 1$ respectively). For the induction step, suppose we know $\phi(Fh(d), z_1)$ and $\phi(Bh(d), z_2)$. We also know, from properties of $Concat$,

$$d = Fh(d) \cdot 2^{\lfloor |d|/2 \rfloor} + Bh(d) \dot{-} 2^{\lfloor |d|/2 \rfloor},$$

so if $z_2 < 2^{\lfloor |d|/2 \rfloor}$ we can conclude $\phi(d, z_2)$ by the basic properties of $Divides$. If on the other hand $z_2 = 2^{\lfloor |d|/2 \rfloor}$, then we can conclude $\phi(d, z_1 \cdot z_2)$, again by the basic properties of $Divides$. pfDCI on the formula $\phi(d)$ assures us that $\phi(Diff(x, y))$, and hence that $BitDiff$ is total.

The fundamental property of $BitDiff$, which we can easily prove without induction (once we know $BitDiff$ is total), is that if $x \neq y$ then

$$Bit(|BitDiff(x, y) \dot{-} 1|, x) \neq Bit(|BitDiff(x, y) \dot{-} 1|, y).$$

An important corollary is

$$x \neq y \leftrightarrow (\exists i \leq |x|)(Bit(i, x) \neq Bit(i, y)).$$

This implies, among other things, that the extensionality axiom used in [Clo89a] is superfluous; note that we have used division in the proof, but only by powers of 2.

Another useful function is $PADSTR(x, y) = x + \max(2^{|x|}, 2^{|y|})$, which takes a binary number x and converts it into a bit-string of length at least $|y| + 1$. This function, defined by composition of previously-defined functions, is obviously total and single-valued. We can prove without induction that $|PADSTR(x, y)| = |PADSTR(y, x)| = 1 + \max(|x|, |y|)$. Furthermore, the $PADSTR$ function can be inverted as follows:

$$UNPADSTR(z) = z \dot{-} 2^{|z| \dot{-} 1}$$

i.e. x is z minus the greatest power of 2 less than or equal to z.

With the aid of these $PADSTR$ and $UNPADSTR$ functions, we can define an invertible pairing function

$$PAIR(x, y) = Concat(PADSTR(x, y), PADSTR(y, x)).$$

Since $PADSTR(x, y)$ and $PADSTR(y, x)$ both have length $1 + \max(|x|, |y|)$, their concatenation has length $1 + 2\max(|x|, |y|)$, so $x = UNPADSTR(Fh(PAIR(x, y)))$ and $y = UNPADSTR(Bh(PAIR(x, y)))$.

We next define two functions $Copies(x, y)$ and $Explode(x, y)$, which also treat their arguments as bit strings. $Copies$, which resembles the $COPY$ function of Part I, produces the concatenation (with $Concat$) of $|x| \dot{-} 1$ copies of y, while $Explode$ produces x's bit-string with $|y| \dot{-} 2$ zeroes inserted after each bit. (The two can be thought of as duals, in a way: each produces $|x| \dot{-} 1$ blocks of $|y| \dot{-} 1$ bits, but $Copies$ gets each block from y while $Explode$ gets each block from a bit of x.) We can prove these total and unique in the system R^1_2. We shall need them in its parameter-free analogue, pfR^1_2, but have been unable to prove their totality in pfR^1_2 for the uninteresting reason that they have more than one parameter, so we can't induct on any one of the parameters unless the rest are universally quantified. Accordingly, for our main theorems we shall augment pfR^1_2 with axioms for their totality.

$$
\begin{aligned}
&Copies(x, y) = z \Leftrightarrow \\
&\quad \Leftrightarrow \quad |z| \dot{-} 1 = (|x| \dot{-} 1) \cdot (|y| \dot{-} 1) \wedge \\
&\qquad\quad (\forall i < |x| \dot{-} 1)(y \dot{-} 2^{|y| \dot{-} 1} = SUBSTR(z, (|y| \dot{-} 1) \cdot i, (|y| \dot{-} 1))) \\
&Explode(x, y) = z \Leftrightarrow \\
&\quad \Leftrightarrow \quad |z| \dot{-} 1 = (|x| \dot{-} 1) \cdot (|y| \dot{-} 1) \wedge \\
&\qquad\quad (\forall i < |x| \dot{-} 1)(Bit(i, x) \cdot 2^{|y| \dot{-} 1} = SUBSTR(z, (|y| \dot{-} 1) \cdot i, (|y| \dot{-} 1)))
\end{aligned}
$$

We prove $Copies$ total in R^1_2 by DCI on x as follows. For $|x| \leq 2$, $Copies(x, y) = y$. For larger values of x, if we assume that both $(\exists z_1)Copies(Fh(x), y) = z_1$ and $(\exists z_2)Copies(Bh(x), y) = z_2$, then $Copies(x, y)$ is simply $Concat(z_1, z_2)$ and so it provably exists.

The proof of totality for $Explode$ is quite similar: for $|x| \leq 2$ we have $Explode(x, y) = 2^{|y|} + Bit(0, x) \cdot 2^{|y| \dot{-} 1}$, while for larger values of x, if we assume $(\exists z_1)Explode(Fh(x), y) = z_1$ and $(\exists z_2)Explode(Bh(x), y) = z_2$, then $Explode(x, y)$ is simply $Concat(z_1, z_2)$ and so it provably exists.

We prove $Explode$ single-valued directly. Suppose $Explode(x, y) = z_1$ and $Explode(x, y) = z_2$, with $z_1 \neq z_2$. Then $BitDiff(z_1, z_2)$ produces a w such that $Bit(|w| \dot{-} 1, z_1) \neq Bit(|w| \dot{-} 1, z_2)$. If $Divides(|y| \dot{-} 1, |w|)$, then let $w' = 2^{\frac{|w|}{|y| \dot{-} 1} \dot{-} 1}$, which has length $\frac{|w|}{|y| \dot{-} 1}$. Then $Bit(|w| \dot{-} 1, z_1)$ is the high bit of $SUBSTR(z_1, (|y| \dot{-} 1) \cdot |w'|, |y| \dot{-} 1)$, while $Bit(|w| \dot{-} 1, z_2)$ is the high bit of $SUBSTR(z_2, (|y| \dot{-} 1) \cdot |w'|, |y| \dot{-} 1)$.

But both of these, by the definition of $Explode$, must be equal to $Bit(|w'| \dot{-} 1, x)$, a contradiction. On the other hand, if $|y| \dot{-} 1$ does *not* divide $|w|$, then $Bit(|w| \dot{-} 1, z_1)$ and $Bit(|w| \dot{-} 1, z_2)$ must both be zero, again a contradiction.

The proof that $Copies$ is single-valued is similar. Suppose $Copies(x, y) = z_1$ and $Copies(x, y) = z_2$, with $z_1 \neq z_2$. Then $BitDiff(z_1, z_2)$ produces a w such that $Bit(|w| \dot{-} 1, z_1) \neq Bit(|w| \dot{-} 1, z_2)$. Letting $w' = 2^{\lfloor \frac{|w|}{|y| \dot{-} 1} \rfloor \dot{-} 1}$, we conclude that $SUBSTR(z_1, |w'| \cdot (|y| \dot{-} 1)), |y| \dot{-} 1)$ and $SUBSTR(z_2, |w'| \cdot (|y| \dot{-} 1)), |y| \dot{-} 1)$ differ in bit $|w| \bmod (|y| \dot{-} 1)$. But adding $2^{|y| \dot{-} 1}$ to each yields y, a contradiction.

The $Explode$ and $Copies$ functions can also be provably inverted on their ranges, in a sense. Define

$$\begin{aligned} Extract\_Copy(x, z) = y &\quad\Leftrightarrow\quad Copies(x, y) = z \\ Implode(y, z) = x &\quad\Leftrightarrow\quad Explode(x, y) = z \end{aligned}$$

These are clearly Σ_1^b definitions, and both functions are provably single-valued in pfR_2^1 (using $BitDiff$ among other things), and provably total in R_2^1. Unfortunately, the one place we need $Implode$ is working inside pfR_2^1; again, we cannot prove it total because it has two parameters, so we augment pfR_2^1 with an axiom for its totality.

Chapter 7

General Induction and How To Simulate It

7.1 General Induction

Suppose S is a partially ordered set with finite descending chains. Let f be a function mapping S to $\mathcal{P}(S)$ such that $(\forall x \in S)(\forall y \in f(x))(y < x)$. Then by the well-foundedness of S, if for all $a \in S$

$$\left(\bigwedge_{y \in f(x)} A(y) \right) \Rightarrow A(a)$$

then A holds everywhere in S. (Note that f of a minimal element is necessarily empty, and other elements may also have no f-images; in these cases the hypothesis reduces to $A(a)$.)

This rule generalizes most forms of mathematical induction. The computational and proof-length distinctions among different forms of induction depend on the size of the sets $f(a)$ and the length of the resulting chains. For example, PIND uses the function

$$f(a) = \begin{cases} \{\lfloor \frac{a}{2} \rfloor\} & \text{if } a > 0 \\ \{\} & \text{otherwise} \end{cases}$$

with S equal to the set of all values represented by terms. LIND uses the function

$$f(a) = \begin{cases} \{a-1\} & \text{if } a > 0 \\ \{\} & \text{otherwise} \end{cases}$$

and restricts S to be the range of the length function. And DCI uses the function

$$f(a) = \begin{cases} \{Fh(a), Bh(a)\} & \text{if } |a| > 2 \\ \{\} & \text{otherwise,} \end{cases}$$

again with S equal to the set of all values represented by terms.

If S and S' are two such posets, with induction functions f_S and $f_{S'}$ respectively, we may be able to simulate S'-induction with S-induction. The idea is to map the induction variable $a' \in S'$ one-to-one to an encoding $ENC(a') \in S$ in such a fashion that for all $a' \in S'$,

$$f_S(ENC(a')) \supseteq \{ENC(y) | y \in f_{S'}(a')\}$$

We shall not require ENC to be surjective, but we require at least that its range be definable by a formula R such that $R(a) \Leftrightarrow a \in \text{Range}(ENC)$. Specifically, for ENC to be useful we shall require a formula $\tilde{A}(a)$, susceptible to S-induction, that is equivalent to $R(a) \supset A(ENC^{-1}(a))$ (provably in the system with S-induction). For example, if S-induction is limited to Σ_i^b-formulæ, we require that $R \in \Pi_i^b$ and $A(ENC^{-1}(a)) \in \Sigma_i^b$ so that $\tilde{A} \in \Sigma_i^b$. (The usual way to ensure $A(ENC^{-1}(a)) \in \Sigma_i^b$ is to make the ENC^{-1} function Σ_1^b-definable on its domain. To do this, in turn, it suffices to Σ_1^b-define ENC, and then to prove within the system that ENC is injective and that R defines its range.) Now $\tilde{A}(a)$ is trivially true if a is not in the range of ENC. On the other hand, if $a \in \text{Range}(ENC)$ and if for all $a' \in S'$ we're given the S' induction hypothesis

$$\left(\bigwedge_{x \in f_{S'}(a')} A(x) \right) \Rightarrow A(a')$$

then we can prove

$$\begin{aligned}
\bigwedge_{y \in f_S(a)} \tilde{A}(y) &\Leftrightarrow \bigwedge_{y \in f_S(a)} (R(y) \supset A(ENC^{-1}(y))) \\
&\Leftrightarrow \bigwedge_{y \in f_S(a) \cap \mathrm{Range}(ENC)} A(ENC^{-1}(y)) \\
&\Rightarrow \bigwedge_{y \in f_{S'}(ENC^{-1}(a))} A(y) \\
&\quad\ \text{since } f_S(ENC(a')) \supseteq ENC(f_{S'}(a')) \\
&\Rightarrow A(ENC^{-1}(a)) \\
&\quad\ \text{by the } S' \text{ induction hypothesis} \\
&\Rightarrow \tilde{A}(a)
\end{aligned}$$

Since this holds for any $a \in S$, we have

$$(\forall x \in S)(\bigwedge_{y \in f_S(x)} \tilde{A}(y) \supset \tilde{A}(x)).$$

We can then apply induction in S to conclude $\tilde{A}$ of an arbitrary term. In particular, if we want to conclude $A(u)$, we use S-induction to prove $\tilde{A}(ENC(u))$, which clearly implies $A(u)$.

7.2 Formal Simulation

In this section we present the skeleton of a sequent-calculus proof of conservation between T_1^i and pfT_1^i, S_1^i and pfS_1^i, R_2^i and pfR_2^i, or the standard and parameter-free forms of any other theory that can define and prove basic properties of suitable encoding functions. In order to give a single, general proof we have abstracted the differences among various forms of induction into metalanguage symbols; one could produce a strict sequent-calculus proof for any given theory and its parameter-free form by replacing these metalanguage symbols with theory-specific language. We first discuss the precise properties necessary for the simulation to work, then state the general form of the conservation theorem (Theorem 30), then give the definitions and commentary necessary to accompany the formal, meta-sequent-calculus proof which is given in Figure 1 (see pages 120 *et seq*). Chapters 8, 9, and 10 then prove the necessary properties of the specific theories pfT^i_k, pfS^i_k, and pfR^i_k and show how to apply Theorem 30 to those theories.

If we restrict our attention to the case of an induction scheme being simulated by its own parameter-free form, we can simplify the notation somewhat. In the general context of Section 7.1, we'll have $S = \mathbf{N}$ under the usual ordering, $S' = \mathbf{N}^{k+1}$ for some k under the usual ordering on the first component, and $f_{S'}(a, b_1, b_2, \dots b_k) = \{(y, b_2, \dots b_k) | y \in f_S(a)\}$. Since $f_{S'}$ is thus easily described in terms of f_S, and S' in terms of S, we shall omit S' and $f_{S'}$ from discussion henceforth and simply let $f = f_S$.

Let $i \geq 1$, let T be a theory of bounded arithmetic, with a fixed set of $\Sigma^b_1 \cup \Pi^b_1$ axioms and f-induction applicable to Σ^b_i-formulæ, and let S_0 be the set of numbers whose f-images are empty, i.e. the base cases of the induction. (For example, S_0 for IND or PIND induction would be $\{0\}$, and for DCI it would be $\{0, 1, 2, 3\}$.) For simplicity we assume that S_0 is finite, and in fact represented by a fixed, finite set of closed terms $s_1, s_2, \dots s_j$. We likewise assume that $f(x)$ is finite for all x, and that in fact there is a fixed, finite set of terms $t_1(x), t_2(x), \dots t_k(x)$ that includes $f(x)$ for all x. (These assumptions exclude certain forms of induction such as "strong mathematical induction", in which $f(x) = \{y \in \mathbf{N} : \mathbf{y} < \mathbf{x}\}$. It seems likely such forms of induction could still be handled so long as the number of predecessors $t_i(x)$ to a given object x, and the number of S_0 elements below a given object x, were reasonable functions of x.) Thus membership in S_0, or in $f(x)$, is represented by a quantifier-free formula; such intuitive notation as $a \notin S_0$ or $t \in f$ conceals no quantifiers, and we shall use it freely. Then T's induction inference can be written in the form

$$\frac{\Gamma(\vec{b}), a \notin S_0, \bigwedge_{t \in f} A(t(a), \vec{b}) \quad \longrightarrow \quad A(a, \vec{b}), \Delta(\vec{b})}{\Gamma(\vec{b}), \bigwedge_{s \in S_0} A(s, \vec{b}) \quad \longrightarrow \quad A(u(\vec{b}), \vec{b}), \Delta(\vec{b})}$$

and the strongly parameter-free analogue in pfT is

$$\frac{a \notin S_0, \bigwedge_{t \in f} A(t(a)) \quad \longrightarrow \quad A(a)}{\bigwedge_{s \in S_0} A(s) \quad \longrightarrow \quad A(u(\vec{b})).}$$

We need an encoding function ENC such that

$$a \notin S_0 \Rightarrow (f(ENC(a,\vec{b})) \supseteq ENC(f(a),\vec{b})).$$

(In practice we shall assume the slightly stronger condition that, for each $t \in f$,

$$a \notin S_0 \Rightarrow t(ENC(a,\vec{b})) = ENC(t(a),\vec{b}).$$

This is inessential, but makes the proofs neater.) For example, to simulate T^i_k with pfT^i_k we need

$$a > 0 \Rightarrow (ENC(a,\vec{b}) - 1 = ENC(a-1,\vec{b})).$$

In at least this case, then, for each $\vec{b}$ the encodings of $\{(a,\vec{b})|a \in \mathbf{N}\}$ must form a contiguous block of natural numbers. Since there are infinitely many $\vec{b}$'s to encode, and the contiguity requirement prevents us from interleaving the encodings for different $\vec{b}$'s, each block must be finite; say, $\{(a,\vec{b})|a < m\}$ for some fixed $m, \vec{b}$. The simplest way to achieve this is to make m actually one of the parameters, and define S' to be only those tuples $(a,\vec{b},m)$ such that $\max(a,\vec{b}) < m$. If we make ENC total and one-to-one on this domain, the proof should go through, since for any given values of u and $\vec{b}$, there is an m such that $(u,\vec{b},m) \in S'$. (This approach is essentially due to Kaye [Kay87], albeit in different notation.) While we have presented this method in order to deal with the theories T^i_k, it proves equally useful for the theories S^i_k and R^i_k.

Like Kaye, we shall actually combine two different encoding functions, each definable in pfT: a surjective encoding of arbitrary k-tuples $\langle b_1, b_2, \ldots b_k\rangle$, and a triple-encoding $[a,b,m]$ with certain other properties, as well as a Π^b_i formula $R(w)$ that provably defines the range of the latter. Neither $\langle\vec{b}\rangle$ nor $[a,b,m]$ is really a function in the strict sense: the former is an infinite family of k-ary functions $\langle\vec{b}\rangle_k$, one for each value of k, while the latter is not necessarily total, and instead we treat $d = [a,b,m]$ as an abbreviation for a certain relation on a,b,m,d which has many of the properties of an injective function. In practice we shall construct the tuple-encoding function by nesting a pairing function, but we state the following definition in the more general setting.

Definition 40 *The "functions" $\langle \vec{b} \rangle$ and $[a, b, m]$ and the relation $R(w)$ are suitable for simulating T in pfT if*

(1) *For every closed term t in the language of T, there is a finite set of terms $n_1, \ldots n_r$ such that $pfT \vdash a \leq t \longrightarrow a = n_1, a = n_2, \cdots a = n_r$*

(2) *There is a Π_i^b-formula defining relation $R(w)$. (Henceforth we shall equate $R(w)$ with this formula defining it.)*

(3) *For each $k \in \mathbf{N}$, the theory pfT Σ_1^b-defines the k-ary function $\langle b_1, b_2, \ldots b_k \rangle$.*

(4) *For each $k \in \mathbf{N}$, $pfT \vdash \longrightarrow (\exists! x_1, x_2, \ldots x_k)(\langle \vec{x} \rangle = d)$*

(5) $pfT \vdash d = \langle \vec{b} \rangle \longrightarrow \max(\vec{b}) \leq d$

(6) *The theory pfT Δ_1^b-defines the 4-ary relation $d = [a, b, m]$.*

(7) *For some term $E(x, y)$, $pfT \vdash \longrightarrow (\exists z)(E(a, b) = [a, b, z])$*

(8) $pfT \vdash d = [a_1, b_1, m_1], d = [a_2, b_2, m_2] \longrightarrow a_1 = a_2 \wedge b_1 = b_2 \wedge m_1 = m_2$

(9) *For some term $B(d)$, $pfT \vdash d = [a, b, m] \longrightarrow \max(a, b, m) \leq B(d)$*

(10) $pfT \vdash s \in S_0, s = [a, b, m] \longrightarrow a \in S_0$

(11) *For all terms $t \in f$, $pfT \vdash d = [a, b, m], a \notin S_0 \longrightarrow t(d) = [t(a), b, m]$*

(12) $pfT \vdash R(s) \longrightarrow (\exists x, y, z)(s = [x, y, z])$

(13) $pfT \vdash s = [a, b, m] \longrightarrow R(s)$

Then we have the

Theorem 30 *Suppose $i \geq 1$, T is a theory of bounded arithmetic with some form of induction applying to all and only Σ_i^b-formulæ, pfT is its parameter-free variant, and there are functions $\langle \vec{b} \rangle$ and $[a, b, m]$ and relation $R(w)$ suitable for simulating T in pfT. If $\Gamma, \Delta \subseteq \Sigma_i^b \cup \Pi_i^b$, and $T \vdash \Gamma \longrightarrow \Delta$, then $pfT \vdash \Gamma \longrightarrow \Delta$. Furthermore, the resulting parameter-free proof can be taken to contain no formulæ not in $\Sigma_i^b \cup \Pi_i^b$.*

Corollary 31 *pfT and T prove the same $\forall\exists\Sigma_i^b$ sentences.*

Proof of Corollary: If $T \vdash \forall\vec{x}\exists\vec{y}\phi(\vec{x},\vec{y})$, where $\phi \in \Sigma_i^b$, then T actually proves $\exists\vec{y}\phi(\vec{a},\vec{y})$ for $\vec{a}$ free variables. So by Parikh's theorem there is a bound $t(\vec{a})$ such that $T \vdash (\exists\vec{y} \leq t(\vec{a}))\phi(\vec{a},\vec{y})$. Since $\phi \in \Sigma_i^b$, this formula too is in Σ_i^b, so by Theorem 30 $pfT \vdash (\exists\vec{y} \leq t(\vec{a}))\phi(\vec{a},\vec{y})$. $(\exists\vec{y})\phi(\vec{a},\vec{y})$ is a weaker statement, so pfT proves it too, and by a $\forall : right$ inference $pfT \vdash \forall\vec{x}\exists\vec{y}\phi(\vec{x},\vec{y})$. ∎

Proof of Theorem 30: Take a proof P of $\Gamma \longrightarrow \Delta$ in T. By the cut-elimination theorem (Theorem 26) we can assume without loss of generality that P is free-cut-free, and hence that every formula in P is either

- (a substitution instance of) a subformula of the endsequent, and hence in $\Sigma_i^b \cup \Pi_i^b$,
- descended directly from an axiom, and hence quantifier-free, or
- descended directly from a principal formula in an induction inference, and hence Σ_i^b.

The only possible non-Σ_i^b formulæ in the proof are Π_i^b subformulæ of the endsequent, and in any case everything is $\Sigma_i^b \cup \Pi_i^b$. It remains to replace each induction inference in the proof, one by one, with an equivalent chunk of proof that contains only parameter-free induction and introduces no new formulæ not in $\Sigma_i^b \cup \Pi_i^b$.

We use a technique based on that used in [Kay87], but abstracted to apply to different forms of induction and formalized in sequent calculus; as a side effect of the latter difference, our results apply to *strongly* parameter-free systems, not just to those with parameter-free axioms as in [Kay87]. As we shall frequently want to refer to an indexed list of formulæ, we define the notation $\{\phi_i\}_{i\in\mathcal{I}}$ to mean the comma-delimited list of all formulæ ϕ_i where i runs over the elements of $\mathcal{I}$. (Counting structural inferences has no bearing on our proofs, so we shall blur the distinction between lists and sets of formulæ.) Let

$$\frac{\Gamma(\vec{b}), a \notin S_0, \{\theta(t(a),\vec{b})\}_{t\in f} \quad\longrightarrow\quad \theta(a,\vec{b}), \Delta(\vec{b})}{\Gamma(\vec{b}), \{\theta(s,\vec{b})\}_{s\in S_0} \quad\longrightarrow\quad \theta(u(\vec{b}),\vec{b}), \Delta(\vec{b})}$$

be an induction inference in proof P, with $\theta \in \Sigma_i^b$ and $\Gamma, \Delta \subseteq (\Sigma_i^b \cup \Pi_i^b)$. By inserting a few $\neg$ inferences before the induction inference we can move individual formulæ in $\Gamma \cup \Delta$ to the opposite side of the sequent arrow until all the formulæ are Σ_i^b. A few more $\neg$ inferences after the induction inference will restore Γ and Δ to their original values, modulo double negations; a few cuts on Σ_i^b-formulæ eliminate the double negations. So without loss of generality we'll assume $\Gamma, \Delta \subseteq \Sigma_i^b$. Then define the formula θ' by

$$\theta'(a, \vec{b}) \Leftrightarrow (\theta(a, \vec{b}) \wedge \bigwedge_{\gamma \in \Gamma} \gamma(\vec{b})) \vee \bigvee_{\delta \in \Delta} \delta(\vec{b})$$

It is easily (i.e. without induction) proven that

$$\Gamma(\vec{b}), \theta(a, \vec{b}) \longrightarrow \theta'(a, \vec{b})$$

and

$$\theta'(a, \vec{b}) \longrightarrow \theta(a, \vec{b}), \Delta(\vec{b}).$$

We can combine these facts with the given induction hypothesis

$$\Gamma(\vec{b}), a \notin S_0, \{\theta(t(a), \vec{b})\}_{t(a) \in f(a)} \longrightarrow \theta(a, \vec{b}), \Delta(\vec{b})$$

to get a corresponding induction hypothesis for θ' without side formulæ

$$a \notin S_0, \{\theta'(t(a), \vec{b})\}_{t(a) \in f(a)} \longrightarrow \theta'(a, \vec{b}).$$

It now remains only to simulate induction on θ', i.e.

$$\frac{a \notin S_0, \{\theta'(t(a), \vec{b})\}_{t(a) \in f(a)} \quad \longrightarrow \quad \theta'(a, \vec{b})}{\{\theta'(s, \vec{b})\}_{s \in S_0} \quad \longrightarrow \quad \theta'(u, \vec{b})}$$

Since $\theta' \in \Sigma_i^b$, we can write $\theta'(a, \vec{b})$ in the form

$$(\exists \vec{z} \leq v(a, \vec{b}))\psi(a, \vec{b}, \vec{z}),$$

where $\psi \in \Pi_{i-1}^b$. Furthermore, the equivalence of θ' with this prenex form can be proven free-cut-freely and without induction.

If $\vec{z}$ is a tuple of variables, or $\vec{c}$ a tuple of constants, such that $\psi(a, \vec{b}, \vec{z})$ and $\psi(a, \vec{b}, \vec{c})$ make sense, we use the notation $\vec{Z}$ or $\vec{C}$ henceforth to denote a tuple of tuples $\vec{z}_s$ or $\vec{c}_s$, one for each element $s \in S_0$. Now let $\chi(a, \vec{b}, \vec{C})$ be the formula

$$\bigwedge_{s \in S_0} \psi(s, \vec{b}, \vec{c}_s) \supset \theta'(a, \vec{b}),$$

which is still in Σ_i^b. It is easily proven (without induction) that

$$\chi(a, \vec{b}, \vec{C}), \{\psi(s, \vec{b}, \vec{c}_s)\}_{s \in S_0} \longrightarrow \theta'(a, \vec{b}).$$

Finally, let $\phi(n)$ be the Σ_i^b-formula

$$(\exists x, \vec{y}, \vec{Z}, w \leq n)(n = [x, \langle \vec{y}, \vec{Z} \rangle, w] \wedge \chi(x, \vec{y}, \vec{Z})).$$

With the aid of a few equality axioms and part 8 of Definition 40, pfT can prove

$$d = [a, \langle \vec{b}, \vec{C} \rangle, m], \phi(d) \longrightarrow \chi(a, \vec{b}, \vec{C})$$

and

$$d = [a, \langle \vec{b}, \vec{C} \rangle, m], \chi(a, \vec{b}, \vec{C}) \longrightarrow \phi(d).$$

(At this point the reader is advised to turn to Figure 1, p. 120, for which the remainder of the chapter is commentary.)

A simple substitution instance of the former fact is

$$t(d) = [t(a), \langle \vec{b}, \vec{C} \rangle, m], \phi(t(d)) \longrightarrow \chi(t(a), \vec{b}, \vec{C}),$$

and we combine this with part 11 of Definition 40 to get sequent **(1)** of Figure 1. Note that the "cut" inference eliminating $\chi(a, \vec{b}, \vec{C})$ is free, but involves only the Σ_i^b-formula χ and therefore does not violate the requirement that the proof must contain only Σ_i^b-formulæ.

We likewise combine the latter fact with the induction hypothesis

$$a \notin S_0, \{\theta'(t(a), \vec{b})\}_{t(a) \in f(a)} \longrightarrow \theta'(a, \vec{b})$$

and the definitions of ψ, χ, and ϕ to yield

$$d = [a, \langle \vec{b}, \vec{C} \rangle, m], \{\chi(t(a), \vec{b}, \vec{C}\}_{t \in f} \longrightarrow \phi(d),$$

which is sequent **(2)** of Figure 1. Note again that the only cuts we have done are Σ_i^b.

Next we cut sequent **(2)** with several instances of sequent **(1)**, one instance for each term t in the "predecessors function" f, to yield sequent **(3)** of Figure 1.

A series of Σ_1^b cuts combines instances of Property 11 with Property 13 to produce sequent **(4)** of Figure 1. This is then combined with sequent **(3)** in a series of $\supset$: $left$ inferences to yield sequent **(5)**.

The formula χ was introduced to bridge the gap between $\{\theta'(s,\vec{b})\}_{s\in S_0}$, which is the base case of an ordinary, non-parameter-free induction, and $\{\phi(s)\}_{s\in S_0}$, which will be the base case of our parameter-free induction. In particular, as shown in the next two chunks of Figure 1, χ holds whenever its first argument is in S_0, and therefore ϕ holds whenever its argument encodes a tuple whose first component is in S_0 (sequents **(6)** and **(7)**).

Thus sequent **(5)** gives sufficient conditions for $\phi([a,\langle\vec{b},\vec{C}\rangle,m])$ when $a\notin S_0$, and sequent **(7)** assures us it holds when $a\in S_0$. The two together form a proof by cases of

$$d=[a,\langle\vec{b},\vec{C},m],\bigwedge_{t\in f}(R(t(d))\supset\phi(t(d)))\longrightarrow\phi(d).$$

Since the particular values $a,\vec{b},\vec{C},m$ are no longer used for anything, we can existentially quantify them and eliminate this assumption with the aid of property 12. A $\supset$: $right$ inference and a $weak:left$ inference produce sequent **(8)**, which is precisely the hypothesis necessary to induct on the formula $R(d)\supset\phi(d)$.

Another application of sequent **(6)**, combined with properties 4, 10, and 12 and the definition of ϕ, tells us that $\phi(s)$ holds whenever s itself is in both S_0 and the range of the encoding function $[\cdots]$. This implies sequent **(9)**, the base case of an induction on $R(s)\supset\phi(s)$.

Now we apply parameter-free induction to sequent **(8)**, choosing to induct up to the term $E(u,\langle\vec{b},\vec{C}\rangle)$, and cut away the base case with sequent **(9)** to produce sequent **(10)**,

$$R(E(u,\langle\vec{b},\vec{C}\rangle))\supset\phi(E(u,\langle\vec{b},\vec{C}\rangle)).$$

But R simply states that its argument is in the range of the encoding function

$[\cdots]$, and the term E was explicitly constructed to produce a value in this range. In the next two chunks of Figure 1 we use these facts to eliminate the assumption $R(E(u, \langle \vec{b}, \vec{C} \rangle))$ and conclude $\phi(E(u, \langle \vec{b}, \vec{C} \rangle))$, which is sequent **(12)**.

By properties 5 and 6, a few equality axioms, and the definitions of χ and ϕ, we conclude that this condition combined with $\bigwedge_{s \in S_0}(\psi(s, \vec{b}, c_s)$ would imply $\theta'(u, \vec{b})$ (see sequent **(13)**). But for each $s \in S_0$, $\psi(s, \vec{b}, c_s)$ follows from $\theta'(s, \vec{b})$. Thus given $\theta'(s, \vec{b})$ for all $s \in S_0$ we can validly conclude $\theta'(u, \vec{b})$; we have effectively simulated full, non-parameter-free induction on θ'. It remains only to cut this result against

$$\Gamma(\vec{b}), \theta(a, \vec{b}) \longrightarrow \theta'(a, \vec{b})$$

and

$$\theta'(u, \vec{b}) \longrightarrow \theta(u, \vec{b}), \Delta(\vec{b})$$

to reach the desired conclusion, in which we have simulated full, non-parameter-free induction on θ with side formulæ Γ and Δ.

Dummy text. More dummy text.

Chapter 8

Conservation for T^i_k

We now prove the first of three specific instances of Theorem 30, that for all $i \geq 1$ and $k \geq 1$, the theory T^i_k is $\forall\exists\Sigma^b_i$-conservative over pfT^i_k. It suffices to define suitable tuple-encoding functions and prove in pfT^1_1 the properties listed in Definition 40, for then those properties will also hold in all other theories pfT^i_k. Note first that property 1 of Definition 40 holds for pfT.

Kaye [Kay87] defines E_1 to be the set of bounded, existential formulæ in the language $\{0, 1, +, \cdot, =, \leq\}$, and the theories IE_1 and IE_1^- to be a certain set of defining axioms augmented by the E_1-IND (respectively, E_1-pfIND) axiom. Chapter 5 of [Kay87] presents two pairing functions corresponding to our $\langle b_1, \ldots b_k\rangle$ and $[a, b, m]$ in order to prove conservation between IE_1^- and IE_1. The same encodings would work to prove conservation between pfT^1_1 and T^1_1, but we shall define slightly different ones that meet the more general requirements of Definition 40.

Definition 41 *If $x_1, x_2, \ldots x_k$ are natural numbers, we define an encoding of k-tuples by nesting the $PAIR$ function of Section 6.10.*

$$\langle x_1, x_2, \ldots x_k\rangle = PAIR(x_1, PAIR(x_2, \cdots PAIR(x_{k-1}, x_k)) \cdots) \dot{-} 1$$

Definition 42 *If x, y, z are natural numbers with $\max(x, y) \leq z$, then we define*

$$[x, y, z] = \frac{z(z+1)(2z+1)}{6} + (z+1)y + x$$

Formally, the formula $[x, y, z] = w$ *will be treated as an abbreviation of*

$$x < z \wedge y < z \wedge 6w = z(z+1)(2z+1) + 6zy + 6x.$$

Note that the above formula is quantifier-free, and hence certainly constitutes a Δ_1^b-definition of the relation $[x, y, z] = w$, which therefore satisfies property 6 of Definition 40.

Since this triple-encoding function is defined whenever $\max(x, y) \leq z$, we define

$$E(x, y) = \frac{\max(x, y)(\max(x, y) + 1)(2\max(x, y) + 1)}{6} + (\max(x, y) + 1)y + x,$$

and this term clearly satisfies part 7 of Definition 40. Furthermore, this encoding function is surjective, so we can define the range-test $R(w) \Leftrightarrow w = w$, which is quantifier-free (hence certainly Π_i^b) and so it trivially satisfies parts 2 and 13 of Definition 40. To satisfy the rest of the requirements of Definition 40, we prove the

Lemma 32 *The following are provable in* pfT_1^1*:*

(1) $(\forall x_1, x_2, \ldots x_k)(\exists! z)(\langle \vec{x} \rangle = z)$ *(this implies part 3 of Definition 40)*

(2) $(\forall z)(\exists! x_1, x_2, \ldots x_k)(\langle \vec{x} \rangle = z)$ *(this implies part 4)*

(3) $(\forall x_1, x_2, \ldots x_k)(\max(\vec{x}) \leq \langle \vec{x} \rangle)$ *(this implies part 5)*

(4) $(\forall x)(\exists! y)(6y = x(x+1)(2x+1))$

(5) $(\forall w)(\exists! x, y, z)(w = [x, y, z])$ *(this implies parts 8 and 12)*

(6) $(\forall w, x, y, z)(w = [x, y, z] \supset \max(x, y, z) \leq w)$ *(this implies part 9)*

(7) $0 = [0, 0, 0]$ *(this implies part 10)*

(8) $(\forall w, x, y, z)((w = [x, y, z] \wedge x > 0) \supset w - 1 = [x - 1, y, z])$ *(this implies part 11)*

Proof : Properties (1-3) of $\langle\vec{x}\rangle$ are clear from the corresponding properties of $PAIR$, proven in Section 6.10.

For (4), the uniqueness of y is trivial. Its existence, which is described as "easy" for the analogous function in [Kay87], seems to require three inductions: one to prove that either x or $x+1$ is a multiple of 2, a second to prove that either x,$x+1$, or $2x+1$ is a multiple of 3, and a third to prove the existence result itself. All three, however, are straightforward parameter-free inductions on x.

For (5), prove by parameter-free induction on w that there is a maximal z_0 such that $6w \geq z_0(z_0+1)(2z_0+1)$. One can prove without induction that this is the unique z for which $(\exists x, y \leq z)([x,y,z] = w)$ is even possible: any value $z > z_0$ would make $[x,y,z] \geq \frac{z(z+1)(2z+1)}{6} > w$, while any $z < z_0$ would force $\max(x,y) \leq z \leq z_0 - 1$ and hence

$$\begin{aligned}
[x,y,z] &\leq \frac{(z_0-1)z_0(2z_0-1)}{6} + z_0 y + x \\
&\leq \frac{(z_0-1)z_0(2z_0-1)}{6} + z_0(z_0-1) + (z_0-1) \\
&= \frac{(z_0-1)z_0(2z_0-1)}{6} + z_0^2 - 1 \\
&< \frac{2z_0^3 - 3z_0^2 + z_0 + 6z_0^2}{6} \\
&= \frac{z_0(z_0+1)(2z_0+1)}{6} \\
&\leq w, \text{ a contradiction.}
\end{aligned}$$

Then subtracting $\frac{z_0(z_0+1)(2z_0+1)}{6}$ from w and dividing[1] by $z+1$ gives us the unique x and y.

(6-8) are immediate from the definition of $[x,y,z]$.

∎

Thus our encoding functions satisfy all the requirements of Definition 40.

[1]This is the only time we use the full power of division, i.e. with a divisor that may not be a power of 2. Even this use of division could be eliminated by requiring m to be a power of 2 and redefining the encoding function to something like $[a,b,m] = mb + a$.

With these encoding functions in hand, we can apply Theorem 30 and conclude

Theorem 33 *For $i \geq 1$ and $k \geq 1$, if $\Gamma, \Delta \subseteq \Sigma_i^b \cup \Pi_i^b$, and $T_k^i \vdash \Gamma \longrightarrow \Delta$, then $pfT_k^i \vdash \Gamma \longrightarrow \Delta$. Furthermore, the resulting parameter-free proof can be taken to contain no formulæ not in $\Sigma_i^b \cup \Pi_i^b$,*

and the corollary that the theories T_k^i and pfT_k^i prove the same $\forall\exists\Sigma_i^b$ sentences.

Chapter 9

Conservation for S^i_k

The second instance of Theorem 30 we shall prove is

Theorem 34 *For $i \geq 1$ and $k \geq 1$, if $\Gamma, \Delta \subseteq \Sigma^b_i \cup \Pi^b_i$, and $S^i_k \vdash \Gamma \longrightarrow \Delta$, then $pfS^i_k \vdash \Gamma \longrightarrow \Delta$. Furthermore, the resulting parameter-free proof can be taken to contain no formulæ not in $\Sigma^b_i \cup \Pi^b_i$.*

Corollary 35 *For $i \geq 1$ and $k \geq 1$, the theories S^i_k and pfS^i_k prove exactly the same $\forall\exists\Sigma^b_i$ sentences.*

Again, Chapter 7 has done most of the work, and we need only define suitable encoding functions. We shall re-use the function $\langle x_1, x_2, \ldots x_k\rangle$ of Chapter 8, exactly as before, so we still have parts 3, 4, and 5 of Definition 40. But we must redefine the encoding function $[x, y, z]$ to satisfy

$$a > 0 \Rightarrow \left(\left\lfloor \frac{[a, b, m]}{2} \right\rfloor = \left[\left\lfloor \frac{a}{2} \right\rfloor, b, m\right]\right).$$

The obvious way to achieve this is to encode the tuple $[a, b, m]$ as b concatenated with the binary form of a; then chopping off the low bit of the encoding *is* chopping off the low bit of a. This can be done as follows:

$$\begin{aligned}[a, b, m] = d \quad \Leftrightarrow \quad & |b| < m^2 \leq |d| < (m+1)^2 \wedge \\ & d = Concat(b + 2^{\min(m^2, |d|)}, a + 2^{|a|}))\end{aligned}$$

That is, we pad b to length m^2 and concatenate the result with a. (Since in $PIND$ we are interested in all the bits of the induction parameter a, rather than treating it as a bit-string and ignoring the high bit, and since the $Concat$ function will treat it as a bit-string, we first prepend a dummy 1 bit to a.) Note that all the functions used in this definition are Σ_1^b-defined, so $[a, b, m] = d$ is provably equivalent to both a Σ_1^b and a Π_1^b formula, so it satisfies part 6 of Definition 40. We insist the length of b be less than m^2, so $|PAD(b, 2^{\min(m^2,|d|)} \dot{-} 1)| = m^2$. Since the maximum possible length of d is $m^2 + 2m$, the maximum possible length of a is therefore $2m$.

$$\underbrace{\boxed{100\cdots 0b_{|b|-1}\cdots b_0}}_{m^2 \text{ bits}}\underbrace{\boxed{a_{|a|-1}\cdots a_0}}_{\leq 2m \text{ bits}}$$

Inversely, given a value of d we can find the appropriate value of m with the formula

$$GET\_M(d) = m \leftrightarrow m^2 \leq |d| < (m+1)^2$$

Such an m always exists, by an obvious $pfLIND$ on d, and it can be proven unique by showing that the squaring function is increasing. $GET\_M(d) = \left\lfloor \sqrt{|d|} \right\rfloor$.

We can now use this to define the inverse functions of the tuple-encoding function $[a, b, m]$. Let $L(d) = \left\lfloor \frac{d}{2^{\min(GET\_M(d)^2,|d|)}} \right\rfloor$; this fraction's value has length $|d| - m^2$, and it can thus be used to separate the high m^2 bits from the remaining low $|d| - m^2$ bits. Define

$$\begin{aligned} GET\_A(d) &= LSP(d, L(d)) \\ GET\_B(d) &= UNPADSTR(MSP(d, L(d))) \end{aligned}$$

Thus the inverse functions are all Σ_1^b-defined, and all can be proven total and unique within pfS_1^1, and clearly satisfy parts 8 and 9 of Definition 40.

The encoding function $[x, y, z]$ itself is of course not total, since it requires $|x| \leq 2z$ and $|y| < z^2$, but it is total for all z sufficiently large to meet these criteria. We can prove

$$|a| \leq 2m \wedge |b| < m^2 \supset (\exists d)(d = [a, b, m])$$

from the known totality of $Concat$ and $PADSTR$, and thus by letting $E(a, b) = [a, b, \max(|a|, |b| + 1)]$ we can satisfy part 7 of Definition 40. The only value excluded from the range of the function $[x, y, z]$ is 0, so with $R(w) \leftrightarrow w > 0$ we can satisfy parts 2, 12, and 13. The remaining requirements, parts 10 and 11, amount in this case to the provability in pfS_1^1 of the sequents $0 = [a, b, c] \longrightarrow a = 0$ and

$$d = [a, b, c], a > 0 \longrightarrow \lfloor \frac{d}{2} \rfloor = [\lfloor \frac{a}{2} \rfloor, b, c].$$

The former is vacuous because 0 isn't in the range at all. The latter is easy from known properties of $Concat$.

Thus we have satisfied all the requirements of Definition 40, and the theorem follows.

Chapter 10

Conservation for R_k^i

In this chapter we prove the

Theorem 36 *For* $i \geq 1$ *and* $k \geq 2$, *if* $\Gamma, \Delta \subseteq \Sigma_i^b \cup \Pi_i^b$, *and* $R_k^i \vdash \Gamma \longrightarrow \Delta$, *then*

$$pfR_k^i + Tot(Copies, Explode, Implode) \vdash \Gamma \longrightarrow \Delta.$$

Furthermore, the resulting parameter-free proof can be taken to contain no formulæ not in $\Sigma_i^b \cup \Pi_i^b$*,*

where $Tot(Copies, Explode, Implode)$ is a set of axioms stating that the *Copies*, *Explode*, and *Implode* functions of Section 6.10 are total. (Alternatively, we could add *Copies* and *Explode* to the language and include invertibility among their defining axioms, thus effectively defining *Implode*.) In either case, R_2^1 is the natural home for these functions, as they have approximately the same growth rate as # — either could in fact be used to replace # — and as they can be defined in R_2^1, just not in pfR_2^1.

10.1 Outline of the Proof

The proof for R_2^i is more difficult than those for S_1^i or T_1^i. We define a rule of *restricted DCI* and prove that it is just as powerful as ordinary DCI, then prove by the same simulation techniques as in Chapters 8 and 9 that strongly parameter-free

DCI (plus the totality of the *Copies*, *Explode*, and *Implode* functions) can simulate it.

Definition 43 *If Φ is a syntactic class of formulæ,* restricted Φ-DCI *is the inference rule*

$$\frac{\Gamma(\vec{b}), A(Fh(a), \vec{b}), A(Bh(a), \vec{b}) \quad \longrightarrow \quad A(a, \vec{b}), \Delta(\vec{b})}{\Gamma(\vec{b}), A(1, \vec{b}), A(2, \vec{b}), A(3, \vec{b}), Pwr2(|t(\vec{b})| \dot{-} 1) \quad \longrightarrow \quad A(t(\vec{b}), \vec{b}), \Delta(\vec{b})}$$

where $A \in \Phi$ and all free variables appear only as indicated.

That is, restricted DCI is ordinary DCI applicable only to terms with length one greater than a power of 2; these terms are precisely those that represent full binary trees when decomposed with the Fh and Bh functions. In an earlier version of this paper, a complex collection of encoding functions was developed to simulate full DCI with restricted DCI, but as it turns out, there is a much simpler proof of the conservation result.

Definition 44 *For $i, k \in \mathbf{N}$, the theory rR^i_k is the theory of bounded arithmetic with the usual axioms for language $\mathcal{L}_k$ and restricted DCI for $\Sigma^b_{i,k}$ formulæ.*

10.2 R^i_k is equivalent to rR^i_k

Recall from Theorem 25 that the induction schemes DCI, PPIND, LPIND, and LLIND are all equivalent, given the ability to introduce new parameters. Telescoping the proofs $DCI \vdash PPIND$, $PPIND \vdash LPIND$, and $LPIND \vdash LLIND$ into one gives a proof that $DCI \vdash LLIND$, but we modify it slightly for technical reasons:

If we wish to perform LLIND on the Σ^b_i-formula $A(x, \vec{y})$, let $B(x, \vec{y}) \Leftrightarrow A(||x| \dot{-} 2|, \vec{y})$. Momentarily ignoring the side formulæ Γ and Δ, we have

$$B(Bh(a), \vec{b}) \quad \Leftrightarrow \quad A(||Bh(a)| \dot{-} 2|, \vec{b})$$

$$
\begin{aligned}
&\Leftrightarrow \quad A\left(\left|\left\lceil \frac{|a| \dot{-} 1}{2} \right\rceil + 1 \dot{-} 2\right|, \vec{b}\right) \\
&\Leftrightarrow \quad A\left(\left|\left\lfloor \frac{|a|}{2} \right\rfloor \dot{-} 1\right|, \vec{b}\right) \\
&\Leftrightarrow \quad A\left(\left|\left\lfloor \frac{|a| \dot{-} 2}{2} \right\rfloor\right|, \vec{b}\right) \\
&\Leftrightarrow \quad A(||a| \dot{-} 2| \dot{-} 1, \vec{b}) \\
&\Rightarrow \quad A(||a| \dot{-} 2|, \vec{b}) \text{ by the induction hypothesis on } A \\
&\Leftrightarrow \quad B(a, \vec{b})
\end{aligned}
$$

so by applying DCI to B, we can conclude

$$B(0, \vec{b}), B(1, \vec{b}), B(2, \vec{b}), B(3, \vec{b}) \longrightarrow B(4t(\vec{b}), \vec{b}).$$

Since $A(0, \vec{b})$ implies the conjunction of formulæ on the left hand side, and the right hand side implies $A(||t(\vec{b})||, \vec{b})$, we conclude

$$A(0, \vec{b}) \longrightarrow A(||t(\vec{b})||, \vec{b}).$$

Of course, we actually have side formulæ, which are only relevant in applying the induction hypothesis on A above; the conclusion is really

$$\Gamma(\vec{b}), A(0, \vec{b}) \longrightarrow A(||t(\vec{b})||, \vec{b}), \Delta(\vec{b}).$$

Now suppose we were given not full DCI but rather parameter-free or restricted DCI. If we eliminate the extra free variables $\vec{b}$ from the proofs, it becomes clear that parameter-free DCI proves parameter-free PPIND, LPIND, and LLIND. And if we are given only restricted DCI, the conclusion is

$$B(0, \vec{b}), B(1, \vec{b}), B(2, \vec{b}), B(3, \vec{b}), Pwr2(|4t| \dot{-} 1) \longrightarrow B(4t(\vec{b}), \vec{b})$$

and so restricted DCI proves corresponding forms of "restricted PPIND", "restricted LPIND," and "restricted LLIND". In particular, since $|4t(\vec{b})| \dot{-} 1 = |t| + 1$ for $t > 0$, the restricted LLIND rule looks like

$$\frac{\Gamma(\vec{b}), A(a, \vec{b}) \quad \longrightarrow \quad A(a+1, \vec{b}), \Delta(\vec{b})}{\Gamma(\vec{b}), A(0, \vec{b}), Pwr2(|t(\vec{b})| + 1) \quad \longrightarrow \quad A(||t(\vec{b})||, \vec{b}), \Delta(\vec{b})} \ .$$

That is, to prove $A(n, \vec{b})$ by restricted LLIND it suffices to find a term t such that $||t|| = n$ and whose length is a one less than a power of 2. The latter seems like no restriction at all, for there is *always* something of length n which is one less than a power of 2. We now formalize this intuition in pfR_2^1.

We first define a function $GROW(x)$ which yields the smallest number of the form 2^{2^n} that is at least as long as x.

$$\begin{aligned} &GROW(x) = y \Leftrightarrow \\ &\quad \Leftrightarrow \quad (1 < |x| \leq |y| < 2|x| \wedge Pwr2(y) \wedge Pwr2(|y| \dot{-} 1)) \vee (x \leq 1 \wedge y = 1). \end{aligned}$$

Note that $GROW(x)$ depends only on the *length* of x, not its value.

The uniqueness of $GROW$ follows immediately from the form of its definition. We prove its totality by parameter-free induction. Clearly $GROW(0) = GROW(1) = 1$ and $GROW(2) = GROW(3) = 2$, so it suffices to show

$$(\exists y)(GROW(Bh(x) = y)) \Rightarrow (\exists y)(GROW(x) = y).$$

In fact we shall prove more concretely that

$$GROW(Bh(x)) = y \Rightarrow GROW(x) = y^2,$$

as follows: for $|x| > 2$, write $|x|$ in the form $2^n + k + 2$, with $0 \leq k < 2^n$. Then $2^n + 1 < |x| \leq 2^{n+1} + 1 = \left|2^{2^{n+1}}\right|$ and so $GROW(x) = 2^{2^{n+1}}$. Now consider $GROW(Bh(x))$. For $|x| > 3$, i.e. $n > 0$, we know

$$\begin{aligned} |Bh(x)| &= \left\lceil \frac{|x| + 1}{2} \right\rceil \\ &= 2^{n-1} + \left\lceil \frac{k+3}{2} \right\rceil \\ &= 2^{n-1} + \left\lfloor \frac{k}{2} \right\rfloor + 2, \end{aligned}$$

and so $GROW(Bh(x)) = 2^{2^n}$. The $n = 0$ case, on the other hand, implies $|x| = 3$ and hence $GROW(x) = 2^{2^1}$, $|Bh(x)| = 2$, and $GROW(Bh(x)) = 2^{2^0}$, so the same result holds. Thus in either case, $GROW(x) = GROW(Bh(x))^2$ and we're done.

Now if we wish to perform LLIND up to an arbitrary term t, it suffices to perform restricted LLIND up to the term $t' = \lfloor \frac{GROW(4t)}{4} \rfloor$. For if $|t| = 2^n + k$ (with $0 \leq k < 2^n$), then $|4t| = 2^n + k + 2$, so $|GROW(4t)| = 2^{n+1} + 1$ and so $|t'| = 2^{n+1} - 1$, which satisfies both $Pwr2(|t'| + 1)$ and $||t'|| = ||t||$. But by Theorem 25, full LLIND proves full DCI, so combining this with the obvious fact that R^i_k is at least as powerful as rR^i_k, we have proven the

Lemma 37 *For $i \geq 1$ and $k \geq 1$,*

$$rR^i_k \equiv R^i_k.$$

That is, rR^i_k and R^i_k prove exactly the same theorems.

It remains only to show that pfR^1_2 can simulate rR^1_2 in the sense of Chapter 7.

10.3 Parameter-free DCI can simulate restricted DCI

We shall once again use the same encoding $\langle x_1, x_2, \ldots x_k \rangle_k$ for k-tuples (ensuring parts 3,4, and 5 of Definition 40), let b denote $\langle b_1, b_2, \ldots b_k \rangle_k$, and redefine the encoding $[x, y, z]$. We construct the function and its associated range-test R to satisfy the following:

- $(\forall x, y)(Pwr2(|x| \dot{-} 1) \supset (\exists z)(\exists! w([x, y, z] = w))$
- $(\forall w)(R(w) \supset (\exists! x, y, z)([x, y, z] = w))$
- $(\forall w, x, y, z)([x, y, z] = w \supset \max(x, y, z) \leq w)$
- $(\forall w < 4)(\forall x, y, z)(w = [x, y, z] \supset x < 4)$
- $(\forall w, x, y, z)((w = [x, y, z] \wedge |x| > 2) \supset (Fh(w) = [Fh(x), y, z] \wedge Bh(w) = [Bh(x), y, z]))$

In order to satisfy the last requirement, the informational content of a (i.e. the bit-string produced by deleting a's leading 1 bit) must be distributed evenly throughout the bit-string $[a, b, m]$. We can accomplish this by making $[a, b, m]$ the concatenation of $|a| - 1$ equal-length blocks, each containing all of b and one meaningful bit of a, and adding a leading 1 bit to control the length.

In encoding tuples for S_1^i, we built a string of length $m^2 + |a|$, where $m^2 > |b|$. In the R_2^i setting we shall require $|m|$ to be of the form 2^{2^r} and build a string of length $1 + 2^{2^r} \cdot (|a| - 1)$, where $|a| - 1$ is required to be of the form $2^n < 2^{2^r}$, and $|b| < 2^{2^r}$. The encoding will be a bit-string of length $2^{2^r + n}$, the concatenation of 2^n blocks of 2^{2^r} bits each, each block containing a single bit of a and all of b.

In order to construct the E term of Definition 40, we need (among other things) to be able to find a suitable m given values of a and b. We only need concern ourselves, however, with values of a whose length is of the form $2^n + 1$. The desired value of m must have length of the form 2^{2^r}, be at least as long as a, and be strictly longer than b. Since $GROW(2|\max(a, b)|)$ is at least as long as both $2|a|$ and $2|b|$, it is strictly longer than (and hence larger than) either $|a|$ or $|b|$, and is of the form 2^{2^r}; we'd like to use it as the length of m. In order to construct an m of that length, we need to exponentiate, and so we need an *a priori* bound in order to apply *LimExp*. By definition of *GROW*,

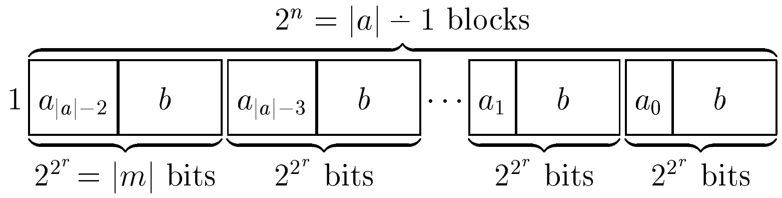
$$
\begin{aligned}
|GROW(2|\max(a, b)|)| &< 2\left|2|\max(a, b)|\right| \\
2^r + 1 &< 2(1 + ||\max(a, b)||) \\
2^r &< 1 + 2||\max(a, b)|| \\
2^{2^r} &< 2 \cdot (2^{||\max(a,b)||})^2 \\
&\leq 2 \cdot (2|\max(a, b)|)^2 \\
&= 8 \cdot |\max(a, b)|^2 \\
2^{2^{2^r}} &< 2^{8|\max(a,b)|^2} \\
&\leq ((2\max(a, b))\#(2\max(a, b)) + 1)^8
\end{aligned}
$$

Anyway, it's bounded by a term in a and b, so let $m = 2^{GROW(2|\max(a,b)|)-1}$, which is long enough to encode a and b (in fact, it's much larger than necessary).

We can now define (the graph of) the encoding function $[a, b, m]$. We say

$$
\begin{aligned}
&[a,b,m] = c \Leftrightarrow \\
&\quad Pwr2(m) \wedge Pwr2(|m|) \wedge Pwr2(||m|| \dot{-} 1) \wedge \\
&\quad Pwr2(|a| \dot{-} 1) \wedge (|a| \leq |m|) \wedge (|b| < |m|) \wedge (|c| \dot{-} 1 = (|a| \dot{-} 1) \cdot |m| \wedge \\
&\quad (\forall i < |a|)(Bit(i \cdot |m| + |m| \dot{-} 1, c) = Bit(i,a) \wedge \\
&\qquad\qquad (\forall j < |m| - 2)(Bit(i \cdot |m| + j, c) = Bit(j,b)))
\end{aligned}
$$

Pictorially, the encoding looks like this:



Such a function can actually be defined in closed form, using the previously-defined functions *Copies* and *Explode*. First we let $B = PADSTR(b, m)$, i.e. b padded to length $2^{2^r} + 1$. Then B consists of a leading 1 bit, some zeroes, and the bits of b; note that since $|b| < 2^{2^r}$, there is at least one zero-bit between the leading 1 and the bits of b. Then $T = Copies(a, B)$ consists of a leading 1 bit followed by 2^n blocks of length 2^{2^r}, each containing at least one leading zero followed by a copy of B. To put the information from a into this string, we simply add $UNPADSTR(Explode(a, B))$, which fills in the 0 bit in each block with a bit of a. The whole thing simplifies to $Copies(a, PADSTR(b, m)) + UNPADSTR(Explode(a, PADSTR(b, m)))$. Replacing m with the term $2^{GROW(2|\max(a,b)|)-1}$ yields a term $E(a, b)$ which satisfies part 7 of Definition 40.

All this can be easily done in R^1_2. Unfortunately, we need to do it in pfR^1_2, and the totality of *Copies* and *Explode* are not obviously provable in pfR^1_2, which is why we need them as axioms to pfR^1_2.

We can likewise define inverse functions for the encoding. Given c, check whether its length is one more than a power of 2; if not, reject it. Otherwise, write

$|c| - 1$ as 2^{n+2^r}, which can be done uniquely with $n < 2^r$. This uniquely determines $|a| = 2^n + 1$ and $|m| = 2^{2^r}$; since we've required m to be a power of 2, m is uniquely determined (although this is inessential). If c was in fact a valid tuple encoding, $\langle\vec{b}\rangle$ is its low $|m| - 1$ bits, and a is the concatenation of a leading 1 bit with the bits in positions $2^n \cdot |m| + |m| - 1, (2^n - 1) \cdot |m| + |m| - 1, \ldots 1 \cdot |m| + |m| - 1, 0 \cdot |m| + |m| - 1$. This last computation is accomplished by the *Implode* function of Section 6.10, which we can prove single-valued but not total in pfR^1_2; again we take its totality as an axiom.

With the ability to decode valid tuple encodings, we know the encoding is injective (part 8 of Definition 40). Directly from the definition we have $\max(a, b, m) \leq [a, b, m]$, so part 9 is satisfied. The set S_0 of base cases for DCI is $\{0, 1, 2, 3\}$, none of which is even in the range of this encoding function, so part 10 is satisfied. The properties $[Fh(x), y, z] = Fh([x, y, z])$ and $[Bh(x), y, z] = Bh([x, y, z])$ are easily proven from the known properties of $Concat$,Fh, Bh, $Explode$, and $Copies$: the string $[x, y, z]$ is a concatenation of a power of 2 equal-length blocks, so its front half is the concatenation of the first half of the blocks, which contain the bits of the front half of x, and its back half is the concatenation of the second half of the blocks, which contain the bits of the back half of x. This proves part 11 of Definition 40.

Finally, the range of the encoding function must be definable by a Π^b_1-formula R. For a number c to be in the range it must first of all have length one more than a power of 2. If it does, let $|c| \dot{-} 1 = 2^{n+2^r}$ with $n < 2^r$, and test whether

$$(\forall i \leq 2^n)(\forall j < 2^{2^r} - 1)(Bit(i \cdot 2^{2^r} + j, c) = Bit(j, c)).$$

In fact, we can construct a closed term, albeit ugly, for all this: $||c| \dot{-} 1| = 1 + n + 2^r$, so $n = UNPADSTR(||c| \dot{-} 1| \dot{-} 1)$ and $r = |||c| \dot{-} 1| \dot{-} 1| \dot{-} 1$. The universal quantifiers turn out to be sharply-bounded, so the resulting formula $R(c)$ is not only Π^b_1 but actually Π_0^b:

$$\begin{aligned} R(c) \quad \Leftrightarrow \quad & Pwr2(|c| \dot{-} 1) \wedge (\exists n, r \leq |c|) \\ & (n = UNPADSTR(||c| \dot{-} 1| \dot{-} 1) \wedge r = |||c| \dot{-} 1| \dot{-} 1| \dot{-} 1 \wedge \\ & (\forall i \leq LimExp(n, |c|))(\forall j < LimExp(LimExp(r, ||c||), |c|) \dot{-} 1) \\ & Bit(i \cdot LimExp(LimExp(r, ||c||), |c|) + j, c) = Bit(j, c). \end{aligned}$$

Thus we have satisfied all the requirements of Definition 40, and the theorem follows.

Bibliography

[All91] William Allen. Arithmetizing uniform NC. *Annals of Pure and Applied Logic*, 53(1):1–50, 1991. see also "Divide and Conquer as a Foundation of Arithmetic", Ph.D. dissertation, University of Hawaii at Manoa, 1988.

[Ara92] Toshiyasu Arai. Frege systems, ALOGTIME and bounded arithmetic. Manuscript, 1992.

[Bar89] D.A. Barrington. Bounded-width polynomial-size branch programs recognize exactly those languages in NC^1. *Journal of Computer and System Sciences*, 38(1):150–164, 1989.

[BC92] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. In Proceedings of the 24th Annual ACM Symposium on the Theory of Computation, pages 283–293. ACM Press, 1992. To appear in *Computational Complexity.*

[Bel92a] Stephen Bellantoni. A new recursion-theoretic characterization of the functional polytime hierarchy. Manuscript, 1992.

[Bel92b] Stephen Bellantoni. *Predicative Recursion and Computational Complexity.* PhD thesis, University of Toronto, 1992.

[BIK+92] Paul Beame, Russell Impagliazzo, Jan Krajíček, Toniann Pitassi, Pavel Pudlák, and Alan Woods. Exponential lower bounds for the pigeonhole principle. In Proceedings of the 24th Annual ACM Symposium on the Theory of Computation, pages 200–220. ACM Press, 1992.

[BIS90] D.A. Mix Barrington, N. Immerman, and H. Straubing. On uniformity within NC^1. *Journal of Computer and System Sciences*, 41:274–306, 1990.

[BK92] Samuel R. Buss and Jan Krajíček. An application of Boolean complexity to separation problems in bounded arithmetic. Submitted for publication, 1992.

[BKT92] Samuel R. Buss, Jan Krajíček, and Gaisi Takeuti. Provably total functions in bounded arithmetic theories R^i_3, U^i_2 and V^i_2. In Peter Clote and Jan

Krajíček, editors, *Arithmetic, Proof Theory and Computational Complexity*. Oxford University Press, 1992. To appear.

[Blo92] Stephen Bloch. Logical analogues to safe recursion. Manuscript in preparation, 1992.

[Bus86] Samuel R. Buss. *Bounded Arithmetic*. Number 3 in Studies in Proof Theory. Bibliopolis (Naples), 1986. see also "The polynomial hierarchy and bounded arithmetic", *Proceedings of the 17th ACM Symposium on Theory of Computing*, 285-290 (1985).

[Bus87] Samuel R. Buss. Polynomial size proofs of the propositional pigeonhole principle. *Journal of Symbolic Logic*, 52:916–927, 1987.

[CKS81] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, January 1981.

[CL90] Kevin Compton and Claude LaFlamme. An algebra and a logic for NC^1. *Information and Computation*, 87:241–263, 1990.

[Clo89a] Peter Clote. A first order theory for the parallel complexity class NC. Technical Report BCCS-8901, Boston College, 1989.

[Clo89b] Peter Clote. Sequential, machine-independent characterizations of the parallel complexity classes $ALOGTIME$, AC^k, NC^k and NC. In Samuel R. Buss and P. Scott, editors, Proceedings of Workshop on Feasible Mathematics. Birkhäuser, 1989.

[Clo92] Peter Clote. On polynomial size Frege proofs of certain combinatorial principles. Manuscript, 1992.

[Cob65] A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, Logic, Methodology and Philosophy of Science II, pages 24–30. North-Holland, 1965. Jerusalem 1964.

[Coo71] Stephen A. Cook. The complexity of theorem proving procedures. In Proceedings of the 3rd Annual Symposium on the Theory of Computing, pages 151–158. ACM Press, 1971.

[Coo75] Stephen A. Cook. Feasibly constructive proofs and the propositional calculus. In Proceedings of the 7th ACM Symposium on the Theory of Computation, pages 83–97, 1975.

[Coo85] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[CT90] Peter Clote and Gaisi Takeuti. Bounded arithmetic and very fast parallel time. Manuscript, 1990.

[CT92] Peter Clote and Gaisi Takeuti. Bounded arithmetic for NC, $ALogTIME$, L and NL. *Annals of Pure and Applied Logic*, 56:73–117, 1992.

[Fer90] Fernando Ferreira. Polynomial time induction. In Samuel Buss and Philip Scott, editors, *Feasible Mathematics*, *Perspectives in Computer Science*, pages 281–291. Birkhäuser, 1990.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische schließen. *Mathematische Zeitschrift*, 39, 1935. English translation in [Sza69].

[GJ79] Garey and Johnson. *A Guide to the theory of $\mathcal{NP}$-completeness.* W.H. Freeman and Company, 1979.

[Hoo83] Jay Hook. *A many-sorted approach to predicative mathematics.* PhD thesis, Princeton University, 1983.

[Imm87] Neil Immerman. Languages that capture complexity classes. *SIAM Journal on Computing*, 16(4):760–778, August 1987.

[Imm89] Neil Immerman. Expressibility and parallel complexity. *SIAM Journal on Computing*, 18(3):625–638, June 1989.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[Kay87] Richard Kaye. *Diophantine and Parameter-free Induction.* PhD thesis, University of Manchester, 1987.

[KPT91] J. Krajíček, P. Pudlák, and G. Takeuti. Bounded arithmetic and the polynomial hierarchy. *Annals of Pure and Applied Logic*, 52:143–153, 1991.

[Kra90] Jan Krajíček. Exponentiation and second order bounded arithmetic. *Annals of Pure and Applied Logic*, 48:261–276, 1990.

[KT90] Jan Krajíček and Gaisi Takeuti. On bounded Σ_1^1 polynomial induction. In *Feasible Mathematics: A Mathematical Sciences Institute Workshop, Ithaca, June 1989*, pages 259–280. Birkhäuser, 1990.

[Lei90] Daniel Leivant. Subrecursion and lambda representation over free algebras. In Samuel Buss and Philip Scott, editors, *Feasible Mathematics*, *Perspectives in Computer Science*, pages 281–291. Birkhäuser, 1990.

[Lei91] Daniel Leivant. A foundational delineation of computational feasibility. In Proceedings of the 6th IEEE Conference on Logic in Computer Science (Amsterdam). IEEE Computer Society Press, 1991.

[Lei92] Daniel Leivant. A foundational delineation of poly-time. *Information and Computation*, 1992. To appear.

[Lin74] J. Lind. Computing in logarithmic space. Technical Report 52, Project MAC, Massachusetts Inst. of Technology, September 1974. 66 pages.

[LM92] Daniel Leivant and Jean-Yves Marion. Purely applicative characterizations of complexity classes. Manuscript, 1992.

[MS72] A.R. Meyer and Larry J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In Proceedings of the 13th Annual IEEE Symposium on Switching and Automata, pages 125–129. IEEE Computer Society Press, 1972.

[Nel86] Edward Nelson. *Predicative arithmetic.* Princeton University Press, 1986.

[Pud85] Pavel Pudlák. Cuts, consistency statements, and interpretations. *Journal of Symbolic Logic*, 50, 1985.

[Ruz81] W. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22:365–383, 1981.

[Sta78] Richard Statman. Bounds for proof-search and speed-up in the predicate calculus. *Annals of Mathematical Logic*, 15:225–287, 1978.

[Sto76] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1976.

[Sza69] M.E. Szabo. *The collected papers of Gerhard Gentzen.* North-Holland, 1969.

[Tak75] Gaisi Takeuti. *Proof Theory.* North-Holland, 1975.

[Tak92] Gaisi Takeuti. *RSUV* isomorphisms. Manuscript, 1992.