A Quantifier-Free String Theory for ALOGTIME Reasoning

by

François Pitt

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Copyright © 2000 by François Pitt
Abstract

A Quantifier-Free String Theory for $ALOGTIME$ Reasoning

François Pitt
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2000

The main contribution of this work is the definition of a quantifier-free string theory $T_1$ suitable for formalizing $ALOGTIME$ reasoning. After describing $L_1$—a new, simple, algebraic characterization of the complexity class $ALOGTIME$ based on strings instead of numbers—the theory $T_1$ is defined (based on $L_1$), and a detailed formal development of $T_1$ is given.

Then, theorems of $T_1$ are shown to translate into families of propositional tautologies that have uniform polysize Frege proofs, $T_1$ is shown to prove the soundness of a particular Frege system $\mathcal{F}$, and $\mathcal{F}$ is shown to provably $p$-simulate any proof system whose soundness can be proved in $T_1$. Finally, $T_1$ is compared with other theories for $ALOGTIME$ reasoning in the literature.

To our knowledge, this is the first formal theory for $ALOGTIME$ reasoning whose basic objects are strings instead of numbers, and the first quantifier-free theory formalizing $ALOGTIME$ reasoning in which a direct proof of the soundness of some Frege system has been given (in the case of first-order theories, such a proof was first given by Arai for his theory $AID$). Also, the polysize Frege proofs we give for the propositional translations of theorems of $T_1$ are considerably simpler than those for other theories, and so is our proof of the soundness of a particular $\mathcal{F}$-system in $T_1$. Together with the simplicity of $T_1$'s recursion schemes, axioms, and rules these facts suggest that $T_1$ is one of the most natural theories available for $ALOGTIME$ reasoning.
Acknowledgements

First and foremost, I would like to thank my advisor, Stephen Cook, without whose help, patience, and guidance this work would have been impossible; Alasdair Urquhart, Alan Borodin, and Charles Rackoff (the other members of my advisory committee) for all their suggestions; and Samuel Buss (my external appraiser) for taking the time to read the whole thesis and saying such nice things about it!

I would also like to thank Stephen Bellantoni and Stephen Bloch, for some interesting and fruitful discussions on $NC$ and $NC^1$; Arnold Rosenbloom, for all the discussions; the staff of the Department of Computer Science, for all their help with administrative matters; and The Fields Institute, for their program on Computational Complexity.

Last, but certainly not least, I would like to thank my parents and my wife, Marie-Josée, for their patience and encouragement over the last six years, as well as for everything else.
Contents

1 Introduction
   1.1 eF-systems and P .................................................. 2
   1.2 F-systems and ALOGTIME ....................................... 3
   1.3 Overview .......................................................... 4

2 The String Algebra \( L_1 \)
   2.1 Basic definitions ..................................................... 7
   2.2 Functions in \( L_0 \) ..................................................... 9
      2.2.1 Basic functions ................................................. 10
      2.2.2 String manipulation functions ............................... 10
      2.2.3 Generalizations of CRN ....................................... 12
      2.2.4 Boolean functions ............................................. 14
   2.3 Functions in \( L_1 \) .................................................. 15
      2.3.1 Basic functions ................................................. 15
      2.3.2 Numerical functions .......................................... 16
   2.4 \( L_1 \) and \( FALOGTIME \) ....................................... 18
      2.4.1 \( FALOGTIME \) is contained in \( L_1 \) ....................... 19
      2.4.2 \( L_1 \) is contained in \( FALOGTIME \) ....................... 20

3 The Quantifier-Free Theory \( T_1 \)
   3.1 Definitions ........................................................ 33
   3.2 Developing the theory ............................................ 36
      3.2.1 Basic definitions and theorems ............................ 36
      3.2.2 Further definitions and theorems .......................... 46
      3.2.3 Numerical definitions and theorems ....................... 57
   3.3 Proving the pigeonhole principle in \( T_1 \) .................... 65
      3.3.1 Representation of PHP in \( T_1 \) ............................. 66
      3.3.2 The \( T_1 \)-proof of PHP ..................................... 68
4 Theorems of $T_1$ Have Polysize $\mathcal{F}$-proofs

4.1 Length functions .......................................................... 73
4.2 Term formulas ............................................................... 74
4.3 Propositional translations ............................................... 75
4.4 The simulation result ...................................................... 75
   4.4.1 Axioms ................................................................... 76
   4.4.2 Rules of inference ....................................................... 80

5 $T_1$ Proves the Soundness of $\mathcal{F}$

5.1 Formalizing $\mathcal{F}$-systems ........................................... 85
   5.1.1 Formulas .................................................................. 85
   5.1.2 Proofs ..................................................................... 87
5.2 Buss’s algorithm for the BSVP .......................................... 87
5.3 Formalizing the BSVP ....................................................... 92
   5.3.1 Proof of correctness in $T_1$ ......................................... 96
5.4 The soundness proof ........................................................ 99
   5.4.1 Preliminaries .............................................................. 99
   5.4.2 The proof .................................................................. 99
5.5 Simulation results ............................................................ 100

6 Related Work

6.1 $T_1$ and $AID$ ................................................................. 103
6.2 $T_1$ and $PV$ ................................................................. 105

7 Conclusion

7.1 Summary ........................................................................ 107
7.2 Future work .................................................................... 108

A Details of Proofs in the Formal Development of $T_1$ 109

Bibliography 135
# Index of Function Symbols

*(Slanted* page numbers indicate $L_1$ definitions, *bold* page numbers indicate $T_1$ definitions, plain page numbers indicate $T_1$ axioms or theorems.)

- $\varepsilon$ (empty string) .................................. 7, 34, 35
- $T^n_k$ (identity) ......................................... 8
- $\cdot$ (concatenation) .................................. 7, 34, 34
- $\sim$ (unary constant) .................................... 9
- $'$ (left bit) ............................................ 10, 38, 38
- $\triangleright$ (left chop) ................................ 10, 34, 34
- $\triangleright$ (left delete) ................................ 10, 37, 37
- $\blacktriangleleft$ (left half) .............................. 8, 34, 35
- $\pi^k_1$ (projection) .................................... 12
- $'$ (right bit) .......................................... 10, 38, 38
- $\blacktriangleright$ (right chop) ............................. 7, 34, 35
- $\blacktriangleright$ (right delete) ......................... 10, 37, 37
- $\blacktriangleleft$ (right half) ............................. 7, 34, 35
- $\#$ (smash) ............................................ 16, 53
- $\times$ (repeat) .......................................... 9
- $\ldots$ (tuple) .......................................... 12
- $0$ (all-zero) .......................................... 10, 34, 35
- $1$ (all-one) .......................................... 10, 34, 35
- $|\cdot|$ (length) ........................................ 16, 60, 61
  - $|\cdot|^N$ ........................................... 18
- CRN .................................................. 8
- $(2^k,2^n)\text{-CRN}$ .................................. 13
- $\text{CRN}_m$ ........................................... 13
- $\ell\text{CRN}_m,\tau\text{CRN}_m$ .............................. 52, 52
- $\ell\text{CRN},\tau\text{CRN}$ ................................ 34, 35
- $\ell\text{powCRN},r\text{powCRN}$ ........................... 56, 56
- left CRN .................................................. 10
- TRN ................................................... 8, 34, 35
- TRN$_k$ .................................................. 15
- $\neg^B,\land^B,\lor^B$, etc. ............................. 14, 48, 49
- not$^B$, and$^B$, or$^B$, etc. ......................... 14, 52
- $\div_k^L,\mod_k^L$ ...................................... 16
- $\div_{2^k}^L,\mod_{2^k}^L$ ................................ 11
- $\max^L$ .................................................. 11, 47
  - $\max_{2^k}^L$ ......................................... 11, 51
- $\min^L$ .................................................. 47
- $\powdiv^L,\powmod^L$ ................................ 54
- $\pow^L$ ................................................. 53
- $\缉^N$ .................................................. 18, 62
- $\prec^N$ ................................................ 17, 57
- $\equiv^N$ ................................................ 17, 57
- $\text{bit}^N$ ............................................. 18
- $\text{carry}^N$ ........................................... 18, 62
- $\text{mask}^N$ ........................................... 18
- $\text{pred}^N$ ............................................ 17
- $\text{succ}^N$ ............................................ 17, 60
- AND .................................................. 16, 49
- CAR, ADD ............................................ 16, 65
- CScar, CSadd .......................................... 15, 64
first_0, first_1 .......................... 18, 61, 62
la_0, la_1 (left adjust) .................... 11
lb (left bit) ............................. 11, 47
lc (left cut) ............................. 11, 47
lp_0, lp_1 (left pad) ...................... 11, 51
OR ........................................ 16, 49
ra_0, ra_1 (right adjust) ................. 11
rb (right bit) ............................. 11, 47
rc (right cut) ............................. 11, 47
rp_0, rp_1 (right pad) .................... 11, 51
sum ........................................ 18, 65
Chapter 1

Introduction

The starting point for this work is the following open problem in complexity theory, concerning propositional proof systems (for a good introduction to propositional proof systems, including the basic definitions, see Cook and Reckhow [19]).

Open Problem 1  Are Frege ("F") and extended Frege ("eF") proof systems p-equivalent?

To provide some motivation for studying Open Problem 1 and to give an indication of its importance, note its connection to some major open questions in complexity theory through the following facts.

General Fact 1  If $NP \neq coNP$, then $P \neq NP$.  

General Fact 2  $NP = coNP$ if and only if $TAUT \in NP$.

General Fact 3  $TAUT \in NP$ if and only if there exists a super (i.e., polynomially-bounded) proof system for $TAUT$.

General Fact 4  Given two proof systems $f_1$ and $f_2$, if $f_1$ is super and $f_2$ p-simulates $f_1$, then $f_2$ is also super.

From Cook and Reckhow’s paper, we know that p-simulation imposes a partial order on proof systems. Determining the relative position of particular proof systems in this order helps shed some light on their relative power and, because of General Fact 4 on such major open problems as $NP \supseteq coNP$ or $P \supseteq NP$. From this point of view, determining the exact position of Frege systems relative to extended Frege systems in this order is one of the most important questions still open in this area. For the rest of this chapter, I will give a short survey of the major results and issues connected with Open Problem 1.
1.1 $e\mathcal{F}$-systems and $P$

It is traditional in complexity theory to equate “feasible” with “polynomial time”. Moreover, there is a close association between polytime and $e\mathcal{F}$-systems since $e\mathcal{F}$-systems can be thought of as reasoning on uniform polysize circuits. For the rest of this work, I will use the traditional notation “$P$” when referring to the class of polytime decidable languages, and “$FP$” when referring to the class of polytime computable functions. Over the years, many characterizations of the classes $P$ and $FP$ have been given, most notably Cobham’s “$\mathcal{L}$” and Bellantoni and Cook’s “$B$”.

- Cobham’s $\mathcal{L}$\cite{16} is the first machine-independent characterization of the class $FP$ using a form of bounded recursion on notation.

- Bellantoni and Cook’s $B$\cite{4} uses a tiered approach (i.e., it distinguishes between “safe” and “normal” parameters) in order to dispense with explicit bounds as in Cobham’s scheme.

Also, many logical theories have been proposed to capture polytime reasoning, most notably Cook’s “$PV$”, Buss’s “$S^1_2$”, Leivant’s “$PT(\mathbb{W})$”, and various second-order theories.

- Cook’s $PV$\cite{18,20} is a free-variable equational theory based on Cobham’s $\mathcal{L}$. Cook showed that every formula $t = u$ provable in $PV$ gives rise to a family of propositional tautologies which assert the equation and have uniform polysize $e\mathcal{F}$ proofs, that $PV$ can define and prove the soundness of $e\mathcal{F}$, and that if the soundness of a propositional proof system $T$ is provable in $PV$, then $e\mathcal{F}$ $p$-simulates $T$.

- Buss’s $S^1_2$\cite{7} is a system of Bounded Arithmetic that can define exactly the polytime functions. Buss showed that $S^1_2$ is $\Sigma^b_1$-conservative over $PV$ (when its language is suitably extended to include all the function symbols of $PV$), which implies that $S^1_2$ proves the soundness of $e\mathcal{F}$ and that the $\Sigma^b_1$-theorems of $S^1_2$ can be translated into propositional tautologies that have uniform polysize $e\mathcal{F}$-proofs (by the corresponding results for $PV$).

- Leivant’s $PT(\mathbb{W})$\cite{26} has generative axioms for $\mathbb{W}$ (intuitively, binary strings) and instances of $\mathbb{W}$-induction as its only axioms. It proves the convergence of exactly the polytime functions over $\mathbb{W}$ (when induction is restricted to positive existential formulas). This formalization is conceptually and technically very simple because it does not rely on any particular initial functions, other than the algebra’s constructors (in fact, the theory can talk about any computable function).

- Buss’s $V^1_1$ (studied by Razborov\cite{27}) and Leivant’s $L_2(QF^+)$\cite{25} are two of the most notable examples of second-order theories for $P$. 
1.2 \( \mathcal{F} \)-systems and \( ALOGTIME \)

The computational power of \( \mathcal{F} \)-systems seems to be captured by the uniform class \( NC^1 \), since \( \mathcal{F} \)-systems can be thought of as reasoning on polysize formulas, which are the same as logarithmic-depth circuits. Recall that \( NC^1 \) is the class of languages decidable by families of logarithmic-depth circuits (\( FNC^1 \) is the functional equivalent, using multi-output circuits), and by results of Ruzzo [28], \( U_{EX} \)-uniform \( NC^1 = ALOGTIME \), where \( ALOGTIME \) is the class of languages decidable in logarithmic time by a random access alternating Turing machine. The functional class \( FALOGTIME \) can be defined in two different ways: if functions are thought of as operating on integers in binary notation, we get a “numerical” version of the class, whereas if functions are thought of as operating on strings of bits (which is closer to the circuit model), we get a “string” version of the class. Fortunately, with a suitable interpretation of numbers as strings (or of strings as numbers), both versions are equivalent.

Therefore, for the rest of this work, I will use \( ALOGTIME \) and \( NC^1 \) interchangeably, always referring to the uniform version of the class (unless otherwise specified). Also, I will use “\( FALOGTIME \)” (or “\( FNC^1 \)”) to refer to the functional version of the class. Various characterizations of \( FALOGTIME \) have been given over the years, most notably Clote’s \( \langle N^0_0 \rangle \) and \( \langle N^0 \rangle \), and Bloch’s string algebra.

- Clote’s \( \langle N^0_0 \rangle \) and \( \langle N^0 \rangle \) [12, 13, 14] are “numerical” characterizations that use restricted forms of Cobham’s recursion on notation. Unfortunately, \( N_0 \) includes a complete function for \( FALOGTIME \) as a base function, and \( N^0_0 \) depends on Barrington’s deep result about bounded-width branching programs [3], so neither algebra is as natural for \( FNC^1 \) as Cobham’s \( L \) is for \( FP \).

- Bloch’s algebra [5, 6] is a “string” characterization that uses the “safe” versus “normal” parameter idea together with a form of recursion similar to Allen’s “divide and conquer recursion” (DCR) [1]. Bloch recognized that Allen’s scheme of DCR (which Allen used to characterize uniform \( NC \)) is particularly well-suited to characterizing uniform parallel complexity classes. Combining this with the tiered approach allows him to dispense with explicit bounds on the rate of growth of functions and to give an elegant characterization that uses only simple base functions and one natural scheme of recursion.

Based on Bloch’s ideas but incorporating some of Clote’s, I will introduce in Chapter 2 a new simple string algebra \( L_1 \) that characterizes \( FALOGTIME \) using very few simple base functions and two simple schemes of recursion (CRN and TRN, to be defined there). It appears to us that \( L_1 \) is simpler than previous characterizations because it has fewer, simpler base functions, and no need for explicit bounds on the growth of functions or for different types of parameters.
Based on the characterizations of $ALOGTIME$ given above, a number of theories to capture $ALOGTIME$ reasoning have been defined, most notably Clote’s “$ALV$” and “$ALV'$”, Takeuti and Clote’s “$TNC^0$”, and Arai’s “$AID$” (all of which are based on “numerical” characterizations of $ALOGTIME$).

- Clote’s $ALV$ and $ALV'$ [13, 14] are free-variable equational theories based on his characterizations of $ALOGTIME$ mentioned above and on Cook’s $PV$. Clote showed that theorems of $ALV$ and $ALV'$ give rise to families of tautologies which have polysize $F$-proofs, but did not show that either of his theories can prove the soundness of $F$-systems. Also, the proof that the propositional translations of theorems of $ALV$ or $ALV'$ have polysize $F$-proofs is fairly involved, and properties of even simple functions (such as “parity” or “majority”) are difficult to prove.

- Takeuti and Clote’s $TNC^0$ [15] (first defined by Takeuti [29]) is a first-order theory similar to Buss’s $S^1_2$ that was shown to be conservative over $ALV'$ (when suitably extended to include every function symbol of $ALV'$). Unfortunately, this theory needs to use a fairly complex form of inference called *bounded successive nomination*, because of its implicit dependence on Barrington’s result (through Clote’s characterization of $ALOGTIME$), which detracts greatly from its simplicity.

- Arai’s $AID$ [2] is a system of bounded arithmetic inspired by Buss’s consistency proof for $F$-systems [9], which proves the soundness of $F$ and whose $\Sigma^b_0$-theorems have polysize $F$-proofs when suitably translated. Moreover, Arai shows that $AID$ is equivalent to a quantified version of Clote’s $ALV$, and hence that $ALV$ can prove the soundness of $F$.

Unlike the situation for $P$, there is no quantifier-free theory for $ALOGTIME$ which has the simplicity and naturalness of $PV$. I claim that $T_1$ fills that role, its axioms and induction schemes being based directly on $L_1$’s simple base functions and natural recursion operations. Moreover, the proofs that propositional translations of the theorems of $T_1$ have uniform polysize $F$-proofs and that $T_1$ can prove the soundness of $F$-systems are much simpler than the corresponding proofs for other theories in the literature.

### 1.3 Overview

Now that I have provided some context and motivation for studying Open Problem 1, let me give a brief overview of the rest of the thesis. In Chapter 2, I will introduce the string algebra $L_1$, followed in Chapter 3 by the quantifier-free theory $T_1$ (including a formal development of the theory, showing how to prove the pigeonhole principle in $T_1$). In Chapter 4, I will define
propositional translations for theorems of $T_1$ and show that they have polysize $\mathcal{F}$-proofs, while in Chapter 5, I will show that $T_1$ proves the soundness of $\mathcal{F}$, by formalizing an algorithm for the “Boolean Sentence Value Problem” (BSVP) in $T_1$, and that $\mathcal{F}$ provably $p$-simulates any proof system whose soundness can be proved in $T_1$. Finally, in Chapter 6, I will compare $T_1$ with various other formalisms for $ALOGTIME$ reasoning, most notably Arai’s $AID$. 
Chapter 2

The String Algebra $L_1$

In this chapter, we define $L_1$ and show that it contains exactly the functions in $\text{FALOGTIME}$. We also give many examples of natural $L_1$ definitions for simple $\text{FALOGTIME}$ functions.

2.1 Basic definitions

The basic objects of the algebra are strings over the alphabet $\{0, 1\}$. The set of all such strings can be defined inductively: $\varepsilon$ (the empty string), 0, 1 are strings, and if $x$ and $y$ are strings, then so is $xy$. Together with a wish for simplicity, this inductive definition motivates our choice of base functions.

The reader should keep in mind that our definitions in this chapter are based on, and guided by, the idea of computation by uniform families of circuits. In particular, all our functions will be length-determined, i.e., the length of a function depends only on the lengths of the arguments, not their values. Also, the starting point for our algebra $L_1$ is Bloch’s paper [6], where he carries out a similar function-algebraic characterization of $\text{FALOGTIME}$, so we will borrow many concepts and definitions from there. (We also borrow certain concepts and definitions from Clote’s work [12, 13, 14].)

Now, we define the base functions and the basic operators that we will use to construct new functions. We use $|x|$ to denote the length of $x$ (i.e., the number of symbols (bits) in the string $x$), $\bar{x}_k$ to denote a $k$-tuple of variables, and $\bar{x}$ to denote an arbitrary tuple of variables.

**BASE:** The set of base functions consists of (in order of increasing arity):

- $\varepsilon, 0, 1 =$ empty string, 0-bit, and 1-bit (constants),
- $\gg x =$ the $\lceil |x|/2 \rceil$ rightmost bits of $x$ (“right half”),
- $x < y = x$ with $|y|$ bits removed from the right (“right chop”),
- $x \cdot y = x$ followed by $y$ (“concatenation”),
Chapter 2. The String Algebra

\[
x ? (y, z_0, z_1) = \begin{cases} 
  y & \text{if } x = \varepsilon, \\
  z_0 & \text{if } x = w \cdot 0 \text{ for some } w, \\
  z_1 & \text{if } x = w \cdot 1 \text{ for some } w,
\end{cases}
\]

(“conditional”)  \[
I^n_k(x_1, \ldots, x_n) = x_k \text{ for any } 1 \leq k \leq n \text{ ("identity" or "projection").}
\]

**Remark 2.1.1** In the definition of \( x ? (y, z_0, z_1) \), it is assumed that \( |z_0| = |z_1| \). If that is not the case, then the value returned will be padded on the left with as many 0’s as are necessary to make \( z_0 \) and \( z_1 \) the same length (the length of \( y \) does not change).

**COMP:** \( f \) is defined from \( g \) and \( h_1, \ldots, h_k \) by composition if
\[
f(\bar{x}) = g(h_1(\bar{x}), \ldots, h_k(\bar{x})).
\]

**CRN:** \( f \) is defined from \( h \) by concatenation recursion on notation on \( x \) if \( h(x, \bar{y}) \in \{0, 1\} \) for all \( x, \bar{y} \) and
\[
f(\varepsilon, \bar{y}) = \varepsilon,
\]
\[
f(x_i, \bar{y}) = f(x, \bar{y}) \cdot h(x_i, \bar{y}) \text{ for } i = 0, 1.
\]

**TRN:** \( f \) is defined from \( g \), \( h \), \( h_\ell \), and \( h_r \) by tree recursion on notation on \( x \) if
\[
f(x, z, \bar{y}) = \begin{cases} 
  g(x, z, \bar{y}) & \text{if } x = \varepsilon, 0, 1, \\
  h(x, z, \bar{y}, f(x_\triangleleft, h_\ell(z), \bar{y}), f(\triangleright x, h_r(z), \bar{y})) & \text{otherwise},
\end{cases}
\]

where \( x_\triangleleft = x < \triangleright \) (the \( \lfloor |x|/2 \rfloor \) leftmost bits of \( x \)). In what follows, we will omit the parameter \( z \) when neither \( g \) nor \( h \) depend on it (in which case the functions \( h_\ell \) and \( h_r \) are irrelevant and will not be specified); we will refer to this form of TRN as simple TRN.

**Remark 2.1.2** Our “right half” function was called “back half (Bh)” by Allen [4] and Bloch [6]. We introduce the new nomenclature because we feel that it is more representative of the action of the function, and the new notation to serve as a graphical reminder of that action (picture the black triangle cutting into the left part of \( x \)). Similarly, our “right chop” function was called “chop” by Cook [20] and “most significant part (Msp)” by Allen and Bloch.

Our new notation should serve as a useful graphical mnemonic for the function’s purpose and action (picture the bits of \( y \) cutting into the bits of \( x \) from the right—in the direction pointed to by the function symbol). Our scheme of CRN is based on the operation of the same name in Clote’s work [12, 13, 14], except that our version has been simplified by eliminating the function
2.2 Functions in \( L_0 \)

\( g \) from the base case (without loss of generality since we can simply concatenate \( g(\vec{y}) \) to the left of our functions to get Clote’s). Our scheme of TRN is based on Bloch’s “very safe DCR”, which is itself based on Allen’s “DCR” (for “divide-and-conquer recursion”), except that our base case is simpler (defined for \( x = \epsilon, 0, 1 \) instead of when \( |x| \leq |b| \) for some extra parameter \( b \)), and we have added the functions \( h_\ell \) and \( h_r \) that allow parameter \( z \) to vary during recursive calls (hence, TRN is technically a scheme of “recursion with replacement”).

**Definition 2.1.1** If we let \( \text{TRN}|_{L_0} \) represent the operation of TRN restricted to functions \( g \in L \) and \( h, h_\ell, h_r \in L' \), for function classes \( L \) and \( L' \), then

- \( L_0 \) is the closure of BASE under COMP and CRN;
- \( L_1 \) is the closure of \( L_0 \) under COMP, CRN, and TRN\( |_{L_0} \), defined recursively.

The next few sections contain mainly function definitions, where the following notational conventions will be used.

- For any constant string \( c \), \( \vec{c}_k \) represents the tuple consisting of \( k \) copies of \( c \).
- Unary functions have higher precedence than binary functions and binary functions have higher precedence than functions of higher arity (keep in mind that “?” has arity 4). Concatenation has higher precedence than any other binary function when represented by juxtaposition; it has lower precedence than any other binary function when represented by “·”.
- \( i \) and \( j \) represent arbitrary fixed single bits, whereas \( k, \ell, m, \) and \( n \) represent arbitrary fixed non-zero natural numbers. When \( 2^k \) is used, \( k \) ranges over all natural numbers (including zero), and similarly for \( 2^\ell \).
- The notation \( k \times x \) stands for \( \overbrace{x \cdots x}^k \) (i.e., \( x \) concatenated with itself \( k \) times). We let \( 0 \times x = \epsilon \) and use \( \hat{k} \) as an abbreviation for \( k \times 1 \), i.e., the unary string representing \( k \).

2.2 Functions in \( L_0 \)

In this section, we define many functions in \( L_0 \) and show that many useful generalizations of CRN can be simulated in \( L_0 \). We are motivated by two goals: to define the machinery necessary to prove that \( L_1 \) contains all of \( FNC^1 \), and to show that many useful functions have simple definitions in our algebra.
2.2.1 Basic functions

First, we define a few simple variations on some of the BASE functions. The rightmost bit of $x$: $x' = x \? (\varepsilon, 0, 1) ; x$ with its rightmost bit removed: $x\ll = x \ll 1$ ; the $\lfloor|x|/2\rfloor$ leftmost bits of $x$: $x\lll = x \lll x$.

Next, a function that reverses the bits of $x$ can be defined by first using CRN to define a function $\text{reverse}(x, y)$, which returns the $|y|$ rightmost bits of $x$ reversed:

\[
\text{reverse}(x, \varepsilon) = \varepsilon,
\]

\[
\text{reverse}(x, yi) = \text{reverse}(x, y) \cdot (x \ll y)' \quad \text{for } i = 0, 1.
\]

Then, $\text{rev}(x) = \text{reverse}(x, x)$ returns the reverse of $x$. Using this function, we can now define symmetric counterparts to some of the earlier functions:

\[
y \triangleright x = \text{rev}(\text{rev}(x) \ll \text{rev}(y)), \quad \triangleright x = \text{rev}(\text{rev}(x) \ll), \quad 'x = \text{rev}(x)'.
\]

Now, let us introduce a generalization of CRN: a function $f$ is defined from $h$ by left CRN (or reverse CRN) on $x$ if $h(x, \vec{y}) \in \{0, 1\}$ for all $x, \vec{y}$ and

\[
f(\varepsilon, \vec{y}) = \varepsilon,
\]

\[
f(ix, \vec{y}) = h(ix, \vec{y}) \cdot f(x, \vec{y}) \quad \text{for } i = 0, 1.
\]

If $f$ is defined from $h$ by left CRN on $x$, then we can use CRN to define

\[
\text{aux}_f(\varepsilon, \vec{y}) = \varepsilon,
\]

\[
\text{aux}_f(x, \vec{y}) = \text{aux}_f(x, \vec{y}) \cdot h(\text{rev}(x), \vec{y}) \quad \text{for } i = 0, 1,
\]

and $f(x, \vec{y}) = \text{rev} \left( \text{aux}_f(\text{rev}(x), \vec{y}) \right)$, using COMP. In what follows, we will use the notational conventions outlined before this section and we will no longer include the trivial base case $f(\varepsilon, \vec{y}) = \varepsilon$ or write “for $i = 0, 1$” when using CRN to define new functions.

2.2.2 String manipulation functions

Now, we will define useful functions for manipulating strings. First, two simple functions that returns a string of the same length as its input, but consisting entirely of 0’s or entirely of 1’s:

\[
j(x) = jx \cdot j \quad \text{(by CRN)}.
\]

Next, we can define a number of functions to compare the lengths of strings (these function symbols will be distinguished by putting a superscript “L” next to them). First, it is useful to have a conditional that tests for the length of a string: $x ?^L (y, z) = x \? (y, z, z)$ is equal
to $y$ if $x$ is empty and equal to $z$ otherwise. Because $\approx_{\mathcal{L}}^2$ distinguishes only between “empty” and “non-empty”, we will define the “length-relational” functions below so that they return $\varepsilon$ when the relation holds and some fixed non-empty string (like “1”) otherwise. Accordingly, we define a simple signum function that returns $\varepsilon$ if its argument is empty and 1 otherwise: $\approx^1 x = x \approx_{\mathcal{L}}^2 (\varepsilon, 1)$ and a corresponding “negation”: $\neg^1 x = x \approx_{\mathcal{L}}^2 (1, \varepsilon)$. Now, we are ready to define the comparison functions.

\[
\begin{align*}
  x \geq^k y & = \approx^k (x \triangleright y) & x \leq^k y & = \approx^k (x \triangleleft y) & x =^k y & = \approx^k ((x \triangleright y) \cdot (x \triangleleft y)) \\
  x >^k y & = \neg^k (x \leq^k y) & x <^k y & = \neg^k (x \geq^k y) & x \neq^k y & = \neg^k (x =^k y) \\
  \max^k(x, y) & = (x \geq^k y) \approx_{\mathcal{L}}^2 (x, y) & \max^k_1(x) & = x \\
  \max^k_{k+1}(x, x_k) & = \max^k(x, \max^k_k(x_k))
\end{align*}
\]

We can also define functions to manipulate the lengths of strings, namely $\text{div}_{2^k}(x)$ that returns a string of 1’s whose length is $\lceil |x|/2^k \rceil$ and a corresponding $\text{mod}_{2^k}(x)$ function satisfying $1x = 2^k \times \text{div}_{2^k}(x) \cdot \text{mod}_{2^k}(x)$.

\[
\begin{align*}
  \text{div}_1^k(x) & = 1x & \text{div}_{2^k}(x) &= \text{div}_k^k(x) \\
  \text{mod}_{2^k}(x) & = (2^k \times \text{div}_{2^k}(x) \triangleright 1x)
\end{align*}
\]

(Interestingly, there does not seem to be a way to define a $\text{div}_k^k$ function for arbitrary $k$ without using TRN.) Following this, we define functions to perform simple bit manipulations on strings (extract single bits or substrings, pad to a certain length).

- “Left bit”: $\text{lb}(x, y) = y \approx_{\mathcal{L}}^2 (\varepsilon, (y \triangleright x))$ returns bit number $|y|$ of $x$ from the left; “right bit”: $\text{rb}(x, y) = y \approx_{\mathcal{L}}^2 (\varepsilon, (x \triangleleft y))$ returns bit number $|y|$ of $x$ from the right (both are equal to $\varepsilon$ if $y = \varepsilon$ or $|y| > |x|$). For convenience, we also define $\text{lb}^0(x, y) = \text{lb}(x, y) ? (0, 0, 1)$ and $\text{rb}^0(x, y) = \text{rb}(x, y) ? (0, 0, 1)$ which return 0 or 1 for all arguments.

- “Left cut”: $\text{lc}(x, y) = x \triangleleft (y \triangleright x)$ returns the $|y|$ leftmost bits of $x$; “right cut”: $\text{rc}(x, y) = (x \triangleleft y) \triangleright x$ returns the $|y|$ rightmost bits of $x$ (both return $\varepsilon$ if $y = \varepsilon$ and $x$ if $|y| \geq |x|$).

- “Left pad”: $\text{lp}_j(x, y) = j(y \triangleleft x) \cdot x$ returns $x$ padded on the left with $j$’s so that $|\text{lp}_j(x, y)| \geq |y|$; “right pad”: $\text{rp}_j(x, y) = x \cdot j(x \triangleright y)$ returns $x$ padded on the right with $j$’s so that $|\text{rp}_j(x, y)| \geq |y|$ (both return $x$ if $|y| \leq |x|$).

- “Left adjust”: $\text{la}_j(x, y) = j(y \triangleleft x) \cdot ((x \triangleleft y) \triangleright x)$ returns $x$ either chopped or padded on the left so that $|\text{la}_j(x, y)| = |y|$; “right adjust”: $\text{ra}_j(x, y) = (x \triangleleft (y \triangleright x)) \cdot j(x \triangleright y)$ returns $x$ either chopped or padded on the right so that $|\text{ra}_j(x, y)| = |y|$.
Finally, we have all the functions we need to define a tuple function \( (\vec{x}_k)_k \) and corresponding projection functions \( (\pi^k_\ell(x)) \). To form tuples, we simply concatenate the arguments together after padding them on the left so that they all have the same length. The projection functions are then defined easily using ◆ and ▶. One small complication arises because we can only divide the length of a string by a power of 2, so we need to form tuples that always have a power of 2 elements even when there are fewer of them that are actually input values. The definitions follow and are inspired by similar definitions in Bloch’s paper [6]. (The tuple function is defined in terms of an auxiliary function \( \text{tuple} \) that has an extra parameter specifying the length to which each value should be padded.)

\[
\text{tuple}_1(x, z) = \text{lp}_0(x, z)
\]

\[
\text{tuple}_{2k}(x_k, y_k, z) = \text{tuple}_{k}(x_k, z) \cdot \text{tuple}_{k}(y_k, z)
\]

\[
\text{tuple}_{2k+1}(x_k, y_{k+1}, z) = \text{tuple}_{k+1}(\varepsilon, x_k, z) \cdot \text{tuple}_{k+1}(y_{k+1}, z)
\]

\[
(\vec{x}_k)_k = \text{tuple}_{k}(\vec{x}_k, \text{max}_k^k(\vec{x}_k))
\]

\[
\pi^k_1(y) = y
\]

\[
\pi^{2k}_\ell(y) = \begin{cases} 
\pi^k_\ell(y) \circ \backslash & \text{if } \ell \leq k \\
\pi^k_{\ell-k} \circ \downarrow y & \text{if } \ell > k 
\end{cases}
\]

\[
\pi^{2k+1}_\ell(y) = \begin{cases} 
\pi^{k+1}_{\ell+1}(y) \circ \backslash & \text{if } \ell \leq k \\
\pi^{k+1}_{\ell-k} \circ \downarrow y & \text{if } \ell > k 
\end{cases}
\]

Note that these functions satisfy the following relations:

\[
\pi^k_\ell((x_1, \ldots, x_k)_k) = \text{lp}_0(x_\ell, \text{max}_k^k(\vec{x}_k)),
\]

\[
\langle \pi^k_1(y), \ldots, \pi^k_{2k}(y) \rangle^{2k}_k = y
\]

(unfortunately, \( \langle \pi^k_1(y), \ldots, \pi^k_k(y) \rangle^k_k \neq y \) for arbitrary \( k \) because of the way the tuple function is defined).

### 2.2.3 Generalizations of CRN

We now introduce a generalization of CRN where the recursion is defined on several variables at once. (We assume that \( x_1, \ldots, x_m \) all have the same length, or are appropriately padded on the left with 0’s to make them all the same length.)

**Definition 2.2.1 (CRN\(_m\))**  We say that \( f \) is defined from \( h \) by \( \text{CRN}_m \) on \( x_1, \ldots, x_m \) if
2.2. Functions in $L_0$

$h(\bar{x}_m, \bar{y}) \in \{0, 1\}$ for all $\bar{x}_m, \bar{y}$ and

\[
f(\bar{x}_m, \bar{y}) = \varepsilon,
\]

\[
f(x_1i_1, \ldots, x_mi_m, \bar{y}) = f(x_1, \ldots, x_m, \bar{y}) \cdot h(x_1i_1, \ldots, x_mi_m, \bar{y}) \quad \text{for } i_1 = 0, 1; \ldots; i_m = 0, 1.
\]

(We can also define left CRN$_m$ similarly to left CRN.) If $f$ is defined from $h$ by CRN$_m$ on $x_1, \ldots, x_m$, then we can define $f$ using CRN as follows: We will define an auxiliary function aux$_f$ by CRN on a parameter $z$; this function will mimic the recursion on $x_1, \ldots, x_m$ by using lc to extract the correct substrings of $x_1, \ldots, x_m$ based on the length of $z$. Then, $f$ is easily defined from aux$_f$ by COMP.

\[
aux_f(zi, \bar{x}_m, \bar{y}) = aux_f(z, \bar{x}_m, \bar{y}) \cdot h(lc(x_1, zi), \ldots, lc(x_m, zi), \bar{y})
\]

\[
f(\bar{x}_m, \bar{y}) = aux_f(max^m_m(\bar{x}_m), lp_0(x_1, max^m_m(\bar{x}_m)) \ldots, lp_0(x_m, max^m_m(\bar{x}_m)), \bar{y})
\]

When a function $f(x, \bar{y})$ is defined by CRN on $x$ from $h$, every bit in the output corresponds to one bit from $x$. Now, we will show how to define a function where every bit of $x$ corresponds to two bits in the output, and then generalize this to arbitrary values (where every group of $2^k$ bits in the input corresponds to a group of $2^n$ bits in the output, which we will call “$2^k$-to-$2^n$-CRN”, or “(2$^k$, 2$^n$)-CRN”).

Following the notation mentioned above, we say that a function $f$ is defined from $h$ by 1-to-2-CRN ($(1, 2)$-CRN) on $x$ if $|h(x, \bar{y})| = 2$ for all $x, \bar{y}$ and

\[
f(\varepsilon, \bar{y}) = \varepsilon,
\]

\[
f(xi, \bar{y}) = f(x, \bar{y}) \cdot h(xi, \bar{y}) \quad \text{for } i = 0, 1.
\]

(We can also define left (1, 2)-CRN.) If $f(x, \bar{y})$ is defined from $h$ by (1, 2)-CRN on $x$, we can define $f$ using CRN as follows: We will first define an auxiliary function aux$_f(z, x, \bar{y})$ by CRN on $z$, to return the $|z|$ leftmost bits of $f(x, \bar{y})$ and then define $f$ from aux$_f$ by COMP. Intuitively, aux$_f(z, x, \bar{y})$ uses div$_2^z(z)$ to determine which bits of $x$ to give as input to $h$ and mod$_2^z(z)$ to determine which bit of $h$ to output next.

\[
aux_f(zi, x, \bar{y}) = aux_f(z, x, \bar{y}) \cdot lb^0(h(lc(x, div^z_2(z) \cdot i), \bar{y}), mod^z_2(z) \cdot i)
\]

\[
f(x, \bar{y}) = aux_f(x \cdot x, x, \bar{y})
\]

Now, we can introduce the generalization mentioned above.

**Definition 2.2.2** ($(2^k, 2^n)$-CRN) We say that $f$ is defined from $g$ and $h$ by 2$^k$-to-2$^n$-CRN ($(2^k, 2^n)$-CRN) on $x$ if $|h(x, \bar{y})| = 2^n$ for all $x, \bar{y}$ and

\[
f(x, \bar{y}) = g(x, \bar{y}) \quad \text{if } |x| < 2^k,
\]

\[
f(x \cdot z, \bar{y}) = f(x, \bar{y}) \cdot h(x \cdot z, \bar{y}) \quad \text{for } z \in \{0, 1\}^{2^k}.
\]
And following Buss [8], we can define functions that implement the boolean connectives in the usual way. Then, \( \approx \) (or \( = \)) rigorously in Chapter 3.

The next functions we will introduce are the Boolean operators, distinguished by putting a superscript \( \text{and} \) next to them. First, we will define a "Boolean test" function, which tests for the truth-value of its argument (where a string’s truth-value is determined by its rightmost bit by convention, with 1 = true and 0 = false — \( \varepsilon \) is treated the same way as 0): \( x ? (y, z) = x ? (z, z, y) \) is equal to \( y \) if \( x \) is “true”; \( z \) if \( x \) is “false” (according to the convention above). Then, \( \approx x = x ? (1, 0) \) returns the truth-value of \( x \) and we can define the boolean connectives in the usual way.

\[
\begin{align*}
\neg^B x &= x ?^B (0, 1) & x \land^B y &= x ?^B (\approx^B y, 0) & x \lor^B y &= x ?^B (1, \approx^B y) \\
x \rightarrow^B y &= x ?^B (\approx^B y, 1) & x \leftrightarrow^B y &= x ?^B (\approx^B y, \neg^B y) & x \oplus^B y &= x ?^B (\neg^B y, \approx^B y) \\
x \geq^B y &= y \rightarrow^B x & x \leq^B y &= x \rightarrow^B y & x \rightarrow^B y &= x \leftrightarrow^B y \\
x <^B y &= \neg^B (x \geq^B y) & x \geq^B y &= \neg^B (x <^B y) & x \neq^B y &= \neg^B (x =^B y)
\end{align*}
\]

Using CRN\(_m\), we can now easily define the following useful functions that perform bitwise operations on their arguments.

\[
\begin{align*}
\text{not}^B (xi) &= \text{not}^B (x) \cdot \neg^B i \\
\text{and}^B (x_1 i_1, \ldots, x_k i_k) &= \text{and}^B (x_1, \ldots, x_k) \cdot (i_1 \land^B \cdots \land^B i_k) \\
\text{or}^B (x_1 i_1, \ldots, x_k i_k) &= \text{or}^B (x_1, \ldots, x_k) \cdot (i_1 \lor^B \cdots \lor^B i_k) \\
\text{xor}^B (x_1 i_1, \ldots, x_k i_k) &= \text{xor}^B (x_1, \ldots, x_k) \cdot (i_1 \oplus^B \cdots \oplus^B i_k) \\
\text{iff}^B (x_1 i_1, \ldots, x_k i_k) &= \text{iff}^B (x_1, \ldots, x_k) \cdot ((i_1 \leftrightarrow^B i_2) \land^B \cdots \land^B (i_{k-1} \leftrightarrow^B i_k))
\end{align*}
\]

And following Buss [8], we can define functions that implement carry-save addition: CScar to compute the carry bits and CSadd to compute the addition bits. Note that these functions are defined so that CScar\((x, y, z, w) + \text{CSadd}(x, y, z, w) = x + y + z + w \) (a fact that will be proved rigorously in Chapter [3])
2.3. Functions in \( L_1 \)

In this section, we define many functions in \( L_1 \) and show that some useful generalizations of TRN can be simulated in \( L_1 \). Again, we are motivated by two goals: to define the machinery necessary to prove that \( L_1 \) contains all of \( FNC^{1} \), and to show that many useful functions have simple definitions in our algebra.

2.3.1 Basic functions

Recall that the operation of TRN is restricted in \( L_1 \) so that we cannot define a function by TRN from functions that are themselves defined by TRN. Hence, it will be useful to be able to define more than one function simultaneously by TRN.

**Definition 2.3.1 (TRN\(_k\))**  The functions \( f_i \) (\( 1 \leq i \leq k \)) are defined from functions \( g_i, h_i, h_\ell, \) and \( h_r \) by TRN\(_k\) on \( x \) if \( |f_1(x,z,\vec{y})| = \cdots = |f_k(x,z,\vec{y})| \) for all \( x,z,\vec{y} \), and for every \( 1 \leq i \leq k \),

\[
f_i(x,z,\vec{y}) = \begin{cases} g_i(x,z,\vec{y}) & \text{if } x = \varepsilon,0,1, \\
                         h_i(x,z,f_1(\bullet, h_\ell(z), \vec{y}), f_1(\cdot x, h_r(z), \vec{y}), \ldots), \\
                         f_k(\bullet x, h_\ell(z), \vec{y}), f_k(\cdot x, h_r(z), \vec{y}, \vec{y}) & \text{otherwise.} \end{cases}
\]

If \( f_k \) are defined from \( g_k, h_k, h_\ell, \) and \( h_r \) by TRN\(_k\), we can define the \( k \)-tuple \( F(x,z,\vec{y}) = (\vec{f}_k(x,z,\vec{y}))_k \) by TRN as follows.

\[
F(x,z,\vec{y}) = \begin{cases} (\vec{g}_k(x,z,\vec{y}))_k & \text{if } x = \varepsilon,0,1, \\
                         (\vec{h}_k(x,z,\pi^k_1(F(\bullet, h_\ell(z), \vec{y})), \pi^k_1(F(\cdot x, h_r(z), \vec{y})), \ldots, \\
                         \pi^k_1(F(\bullet x, h_\ell(z), \vec{y}), \pi^k_1(F(\cdot x, h_r(z), \vec{y}), \vec{y}))_k & \text{otherwise.} \end{cases}
\]

Then, a simple composition gives \( f_i(x,z,\vec{y}) = \pi^k_1(F(x,z,\vec{y})) \) for \( 1 \leq i \leq k \).

Now, we can define some functions by TRN (actually, by simple TRN). The first two perform Boolean operations on all the bits of their input; \( |x| \) returns the length of \( x \), expressed as a binary number; \( x \# y \) returns \( |x| \) copies of \( y \) concatenated together (so that \( |x \# y| = |x| \times |y| \)).
\( \text{div}_k^L(x) \) returns a string whose length is equal to \(|x|/k|\) (where we use the notation “\( \text{mod}_k^L(x) \)” as a shorthand for the term \((k \times \text{div}_k^L(x)) \gg_{1} x\)); and the last two functions are defined by TRN\(_2\) and will be used to count the number of 1’s in the string \(x\), using the carry-save technique of Buss \[8\] (this will be done below).

\[
\begin{align*}
\text{AND}(x) &= \begin{cases} 
    x & \text{if } x = \varepsilon, 0, 1 \\
    \text{AND}(\downarrow x) \land \text{AND}(\uparrow x) & \text{otherwise}
\end{cases} \\
\text{OR}(x) &= \begin{cases} 
    x & \text{if } x = \varepsilon, 0, 1 \\
    \text{OR}(\downarrow x) \lor \text{OR}(\uparrow x) & \text{otherwise}
\end{cases} \\
|\!|x\!| &= \begin{cases} 
    1x & \text{if } x = \varepsilon, 0, 1 \\
    x \ll \gg x ?^{2L} \left( |\!|\downarrow x\!| \cdot 0, |\!|\uparrow x\!| \cdot 1 \right) & \text{otherwise}
\end{cases} \\
\!\!\#\!\!x\!\#\!y &= \begin{cases} 
    x ?^{2L} (\varepsilon, y) & \text{if } x = \varepsilon, 0, 1 \\
    (\downarrow x \# y) \cdot (\uparrow x \# y) & \text{otherwise}
\end{cases}
\end{align*}
\]

\[
\text{div}_k^L(x) = \begin{cases} 
    x \gg \widehat{k} ?^{2L} (\downarrow x, \varepsilon) & \text{if } x = \varepsilon, 0, 1 \\
    \text{div}_k^L(\downarrow x) \cdot \text{div}_k^L(\uparrow x) \cdot ( (\text{mod}_k^L(\downarrow x) \cdot \text{mod}_k^L(\uparrow x)) \gg \widehat{k} ?^{2L} (1, \varepsilon) ) & \text{otherwise}
\end{cases}
\]

\[
\text{CAR}(x) = \begin{cases} 
    0x & \text{if } x = \varepsilon, 0, 1 \\
    \text{CScar}(\text{CAR}(\downarrow x), \text{CAR}(\uparrow x), \text{ADD}(\downarrow x), \text{ADD}(\uparrow x)) & \text{otherwise}
\end{cases}
\]

\[
\text{ADD}(x) = \begin{cases} 
    x & \text{if } x = \varepsilon, 0, 1 \\
    \text{CSadd}(\text{CAR}(\downarrow x), \text{CAR}(\uparrow x), \text{ADD}(\downarrow x), \text{ADD}(\uparrow x)) & \text{otherwise}
\end{cases}
\]

Note that \(|\text{CAR}(x)| = |\text{ADD}(x)|\) (easy to show inductively) so the definition by TRN\(_2\) is correct. Also note that with \(\text{div}_k^L\) and \(\text{mod}_k^L\), we can now define “\((k, \ell)\)-CRN” similarly to \((2^k, 2^\ell)\)-CRN, but for blocks of bits of arbitrary fixed lengths. Interestingly, it does not seem possible to define a more general \(\text{div}^L\) function that would take two parameters, and thus to define a general form of CRN where the lengths of the input and output blocks of bits are specified by extra parameters (we discuss this issue further in Chapter 3).

### 2.3.2 Numerical functions

Unfortunately, the fact that functions in \(L_1\) are length-determined makes it harder to define “numerical” functions, \(i.e.,\) functions that treat their inputs as binary notation for numbers (ignoring leading 0’s). For example, the definition of \(|x|\) given above is quite simple whereas the definition of \(|x|^N\) given below relies on some more complex functions.
2.3. Functions in $L_1$

Now, we will define a number of “numerical” functions (distinguished by putting a superscript “$N$” next to them). We start with an equality operator for numbers, and also one for strings.

$$x =^N y = \text{AND}(1 \cdot \text{iff}_2^B(x, y))$$

$$x =^S y = \neg^B(x =^L y) \land^B (x =^N y)$$

Note that the extra “1” is necessary in the definition of $=^N$ for $\varepsilon =^N \varepsilon$ to be true, and the value of $x =^N y$ is independent of the lengths of $x$ and $y$, i.e., the function really does behave as though its string inputs were binary representations of numbers. Also note that “$\neg^B$” is necessary in the definition of $=^S$ because of our convention that $=^L$ returns $\varepsilon$ for “true” and 1 for “false”.

Next, we define a successor and a predecessor function, both by left CRN. The successor function is defined in terms of an auxiliary function that simply replaces each bit by its negation until it encounters a 0, which it replaces by 1, and then outputs each bit unchanged (e.g., 11010 becomes 11011, 1011 becomes 1100, 111 becomes 000). The successor function first adds a 0 to the front (left) of its argument before calling the auxiliary function, in case the string consists of all 1’s (e.g., 111 correctly becomes 1000 and not just 000). The predecessor function performs a similar computation, except replacing bits by their negation until it encounters a 1, and first checking that the string does not consist of all 0’s before calling the auxiliary function.

$$\text{aux}_\text{succ}^N(ix) = (\text{AND}(1x) ?^B (\neg^B i, i)) \cdot \text{aux}_\text{succ}^N(x) \quad \text{succ}^N(x) = \text{aux}_\text{succ}^N(0x)$$

$$\text{aux}_\text{pred}^N(ix) = (\text{OR}(x) ?^B (i, \neg^B i)) \cdot \text{aux}_\text{pred}^N(x) \quad \text{pred}^N(x) = \text{OR}(x) ?^B (\text{aux}_\text{pred}^N(x), x)$$

Note that one unfortunate side-effect of the fact that the functions are length-determined is that the successor function always appends a bit to the left of its argument. So starting from $\varepsilon$ and applying $\text{succ}^N$ repeatedly, we get a series of strings that represent 0, 1, 2, . . . in binary, but whose lengths are also 0, 1, 2, . . . Next, we define the numerical predicate $<^N$, which together with $=^N$ allows us to define all other relational operators on numbers using the Boolean connectives.

Note that to define $\text{less}^N$, we use Clote’s “programming trick” [12] of making a sweep through the bits of the strings $x$ and $y$, appending a 1 when some condition is met so that the final composition with $\text{OR}$ yields 1 iff the condition was met at some position. Using $\text{AND}$, we could similarly define functions that test for some condition on every bit of their inputs. Also recall that functions defined by CRN$_2$ (such as $\text{less}^N$ below) first pad their arguments on the left with 0’s so they have the same length.

$$\text{less}^N(xi, yj) = \text{less}^N(x, y) \cdot (i <^B j) \land^B (x =^N y) \quad x <^N y = \text{OR}(\text{less}^N(x, y))$$

The next function we want to define is $\text{bit}^N(x, z)$, which returns bit number $z$ of $x$, starting at 1 and counting from the right, where $z$ is interpreted as a binary number. The easiest way to
do this is by defining a function $\text{pow}^N(z, x)$ that returns a string of length $|x|$ consisting entirely of 0’s except at bit position $z$ (from the right), if $1 \leq^N z \leq^N |x|$. Then, we define a function $\text{maskbit}^N(x, y)$ that treats $y$ as a mask to determine which bit of $x$ to return.

$$\text{pow}^N(z, ix) = \left( |ix| =^N z \right) \cdot \text{pow}^N(z, x)$$

$$\text{maskbit}^N(x, y) = \text{OR}(\text{and}^B(x, y))$$

$$\text{bit}^N(x, z) = \text{maskbit}^N(x, \text{pow}^N(z, x))$$

Next, we want to define addition. This will require only a few more definitions. First, in order to simulate a function that strips leading ones (or zeros) from $x$, we can define functions $\text{first}_j(x)$ that return a mask which is 1 on the leftmost bit of $x$ equal to $j$ and 0 elsewhere. We can also define a function that returns a mask which is 1 on every significant bit of $x$ (i.e., every bit to the right of the first “1” in $x$) and 0 elsewhere.

$$\text{first}_1(xi) = \text{first}_1(x) \cdot (\text{OR}(x) ?^B (0, i))$$

$$\text{first}_0(xi) = \text{first}_0(x) \cdot (\text{AND}(1x) ?^B (\neg^B i, 0))$$

$$\text{mask}^N(xi) = \text{mask}^N(x) \cdot (\text{OR}(x) ?^B (1, i))$$

Then, we can define a function which computes the carry bits and an addition function, as follows.

$$\text{carry}^N(ix, jy) = \text{maskbit}^N(\text{and}^B(ix, jy), \text{first}_0(\text{xor}^B(ix, jy))) \cdot \text{carry}^N(x, y)$$

$$x +^N y = \text{xor}^B(\text{carry}^N(x, y) \cdot 0, x, y)$$

Finally, using the addition function, we can define a function that counts the number of ones in a string: $\text{sum}(x) = \text{CAR}(x) +^N \text{ADD}(x)$ and using this function, define the “numerical” length of $x$: $|x|^N = \text{sum}(\text{mask}^N(x))$. (Note that we could also have defined $|x| = \text{sum}(1x)$ instead of directly using TRN.)

### 2.4 $L_1$ and $\text{FALOGTIME}$

In this section, we prove the following claim.

**Claim 2.4.1** $L_1 = \text{FALOGTIME} (= \text{uniform FNC}^1)$.

To be precise, we say that a $k$-ary function $f$ belongs to $\text{FALOGTIME}$ if there exists an integer polynomial $p_f$ such that $|f(\vec{x}_k)| \leq p_f(|x_1|, \ldots, |x_k|)$ for every $\vec{x}_k$, and if the language $\{(\vec{x}_k, i, b) : \text{the } i\text{-th bit of } f(\vec{x}_k) \text{ is equal to } b\}$ is recognizable by an ATm running in time $O(\max\{\log |x_1|, \ldots, \log |x_k|\})$. 


2.4.1 \textit{FALOGTIME} is contained in \( L_1 \)

To prove that \( \text{FALOGTIME} \subseteq L_1 \), we show how to simulate the computation of an \( \text{ATM} \) using functions in \( L_1 \). Then, if \( f \in \text{FALOGTIME} \), there exists a term \( t_f \in L_1 \) such that \( |t_f(\bar{x}_k)| = p_f(|\bar{x}_k|) \), so we can use \( \text{CRN} \) on \( t_f(\bar{x}_k) \) to compute each bit of \( f \) by simulating the \( \text{ATM} \) on the appropriate input. (Technically speaking, this works only if \( |f(\bar{x}_k)| = |t_f(\bar{x}_k)| \) for all \( \bar{x}_k \), but if that is not the case, we can simply use Bloch’s idea of “length masks” to compute a mask \( b_f(\bar{x}_k) \) of length \( t_f(\bar{x}_k) \) that has a 1 in every bit position where \( f(\bar{x}_k) \) is defined and a 0 elsewhere.)

Now, without loss of generality, let the \( \text{ATM} \) have the following properties.

1. There is a function \( t \in L_1 \) such that the \( \text{ATM} \) runs for no more than \( ||t(\bar{x})|| = O(\log |t(\bar{x})|) \) steps on inputs \( \bar{x} \) (always possible when the \( \text{ATM} \) runs in logarithmic time since \( t \) can use \# and \cdot to output a string whose length is an arbitrary polynomial in the lengths of the inputs). Also, universal states of the \( \text{ATM} \) are given even numbers and existential states, odd numbers. Moreover, we assume that the function \( t \) is defined so that \( |t(\bar{x})| \) is always a power of 2 and \( 2^{|t(\bar{x})|} \) is greater than the number of states of the \( \text{ATM} \) for any inputs \( \bar{x} \) (in other words, a string of length \( 2|t(\bar{x})| \) is long enough to encode the state of the \( \text{ATM} \)).

2. The \( \text{ATM} \) has \( n \) read-only input tapes represented by strings \( x_1, \ldots, x_n \) and \( k \) worktapes represented by pairs of strings \( y_1^{\ell}, y_1^r, \ldots, y_k^{\ell}, y_k^r \), each of length exactly \( 2|t(\bar{x})| \), where \( y_i^{\ell} \) represents the content of tape number \( i \) to the left of the tape head and \( y_i^r \) represents the content to the right, with the head scanning the rightmost symbol of \( y_i^{\ell} \). Each of the three possible worktape symbols (1, 0, or blank) is encoded using two bits (11 for 1, 10 for 0, and 00 for blank). Initially, the worktapes are blank.

3. The computation tree of the \( \text{ATM} \) is a complete binary tree (each non-leaf node has exactly two successor configurations, a left successor and a right successor, and every leaf occurs at the same level).

4. Access to the input occurs only at the leaves of the computation tree and is of the form “accept iff symbol number \( y_i^j \) (interpreted as a binary number) on input tape number \( j \) is equal to \( b \)”, where \( i, j, \) and \( b \) are encoded in the current state of the \( \text{ATM} \).

Then, if we let \( \text{CON} = \langle s, y_1^{\ell}, y_1^r, \ldots, y_k^{\ell}, y_k^r \rangle_{2k+1} \) represent a configuration of the \( \text{ATM} \) when in state \( s \), we can define \( \ell \text{CON} \) and \( r \text{CON} \), the left and right successor configurations of \( \text{CON} \), as follows:

\[
\ell \text{CON} = \langle \text{ltape}_i^\ell(\text{CON}), \text{ltape}_i^r(\text{CON}), \ldots, \text{ltape}_k^\ell(\text{CON}), \text{ltape}_k^r(\text{CON}) \rangle_{2k+1},
\]

\[
r \text{CON} = \langle \text{rtape}_i^\ell(\text{CON}), \text{rtape}_i^r(\text{CON}), \ldots, \text{rtape}_k^\ell(\text{CON}), \text{rtape}_k^r(\text{CON}) \rangle_{2k+1},
\]
where each of \( l\text{state}, l\text{tape}_i^e, r\text{tape}_i^e, r\text{state}, r\text{tape}_i^f, r\text{tape}_i^f \) is easily seen to be in \( L_0 \), involving only simple string manipulations and finite table lookup on the state \( s \) (for example, \( l\text{tape}_i^f(\con) = d_i \ ?^\B (00 \triangleright ((y_i^f < 00 \cdot b_i) \cdot \lc(y_i^f, 00)), (00 \cdot y_i^f) < 00) \) computes the contents of tape \( i \) to the left of the head in the left successor of \( \con \), where \( d_i \) (the direction of movement for head \( i \) ) and \( b_i \) (the tape symbol to write on tape \( i \) ) are obtained from the state and tape contents of \( \con \) using the conditional function). Moreover, if we let \( \text{select} \) be defined by \((2,1)\)\(-\text{CRN} \) to output every second bit of its input string, the function \( \text{input}(\con, \vec{x}) = \text{bit}^\B(x_j, \text{select}(y_i^f)) \rightarrow^b b \) (where \( i, j, \) and \( b \) are extracted from the state \( s \) ) is equal to the accept state of the given input configuration and is in \( L_1 \). Finally, we let \( \con_0 = (s_0, t(\vec{x}) \# 00, \ldots, t(\vec{x}) \# 00)_{2k+1} \) denote the initial configuration, where \( s_0 \) is the initial state of the \( \text{ATM} \).

Now, we can easily use TRN to define a function \( \text{eval} \) that evaluates the computation tree of the \( \text{ATM} \), so that the result of the entire computation is given by \( \text{eval}(t(\vec{x}), \con_0, \vec{x}) \):

\[
\text{eval}(z, \con, \vec{x}) = \begin{cases} 
\text{input}(\con, \vec{x}) & \text{if } z = \varepsilon, 0, 1, \\
\pi_1^{2k+1}(\con)' \ ?^B (\text{eval}(z, \ell\con, \vec{x}) \lor^B \text{eval}(\triangleright z, r\con, \vec{x})), \\
\text{eval}(z, \ell\con, \vec{x}) \land^B \text{eval}(\triangleright z, r\con, \vec{x}) & \text{otherwise}.
\end{cases}
\]

Note that in the recursive call, \( \pi_1^{2k+1}(\con) \) simply extracts the current state from the given configuration, and the rightmost bit of the state number is used to determine whether the configuration is universal or existential. Also note that this half of the proof is considerably simpler than the corresponding proofs in Bloch \[6\] and Clote \[13, 14\]. This seems to be because our scheme of TRN encapsulates the sort of computation carried out by \( \text{ATM} \)'s more directly than the schemes considered by Bloch and Clote, especially by its use of the parameter replacement functions \( h_{\ell} \) and \( h_{r} \).

### 2.4.2 \( L_1 \) is contained in \text{FALOGTIME} \n
To prove that \( L_1 \subseteq \text{FALOGTIME} \), we argue that every function in \( L_0 \) can be computed by a family of circuits in uniform \( \text{FNC}^0 \), and that every function in \( L_1 \) can be computed by a family of circuits in uniform \( \text{FNC}^1 \), where we use Bloch’s notion of \text{mapping-uniformity}, defined in \[6\], which generalizes \( U_E \)-uniformity to make sense for circuits of constant depth. As will be seen, the facts that functions in \( L_0 \) have constant depth circuits and that functions in \( L_1 \) have logarithmic depth circuits are quite simple to prove; the technical difficulties arise mainly from uniformity considerations.

First, we give bounds on the rate of growth of functions in \( L_0 \) and \( L_1 \).

**Lemma 2.4.2** For every \( n \)-ary function \( f \in L_0 \), there exist constants \( a_0^f, a_1^f, \ldots, a_n^f \in \mathbb{N} \) such that \( |f(x_1, \ldots, x_n)| \leq a_0^f + a_1^f |x_1| + \cdots + a_n^f |x_n| \) for all strings \( x_1, \ldots, x_n \).
2.4. \( L_1 \) AND FALOGTIME

**Proof**  The result is proved by induction on the definition of \( f \).

- If \( f = \varepsilon \), then \( a_0^f = 0 \) since \(|\varepsilon| = 0\).
- If \( f = 0 \) or \( f = 1 \), then \( a_0^f = 1 \) since \(|0| = |1| = 1\).
- If \( f(x) = x \cdot x \), then \( a_0^f = 0, a_1^f = 1, a_2^f = 0 \) since \(|x \cdot y| = |x| + |y| \leq |x|\).
- If \( f(x, y) = x \cdot y \), then \( a_0^f = 0, a_1^f = 1, a_2^f = 1 \) since \(|x \cdot y| = |x| + |y|\).
- If \( f(x, y, z, w) = x ? (y, z, w) \), then \( a_0^f = 0, a_1^f = 0, a_2^f = 1, a_3^f = 1, a_4^f = 1 \) since \(|x ? (y, z, w)| \leq \max\{|y|, |z|, |w|\} \leq |y| + |z| + |w|\).
- If \( f(x_1, \ldots, x_n) = T_k^n(x_1, \ldots, x_n) \), then \( a_0^f = 0, \ldots, a_{k-1}^f = 0, a_k^f = 1, a_{k+1}^f = 0, \ldots, a_n^f = 0 \) since \(|T_k^n(x_1, \ldots, x_n)| = |x_k|\).
- If \( f \) is defined by CRN from \( h \), then \(|f(x, \overline{y})| = |x| \) so \( a_0^f = 0, a_1^f = 1, a_2^f = 0, \ldots, a_{n+1}^f = 0 \).
- If \( f \) is defined by \( \text{COMP} \) from \( g \) and \( h_1, \ldots, h_k \), then

\[
|f(\overline{x})| = |g(h_1(\overline{x}), \ldots, h_k(\overline{x}))| \\
\leq a_0^g + a_0^g|h_1(\overline{x})| + \cdots + a_k^g|h_k(\overline{x})| \\
\leq a_0^g + \sum_{1 \leq i \leq k} a_i^g \left( a_0^{h_i} + \sum_{1 \leq j \leq n} a_j^{h_i}|x_j| \right) \\
\leq \left( a_0^g + \sum_{1 \leq i \leq k} a_i^g a_0^{h_i} \right) + \sum_{1 \leq i \leq k} \left( \sum_{1 \leq j \leq n} a_j^{h_i}|x_j| \right) \\
\text{so } a_0^f = a_0^g + a_0^g a_0^{h_1} + \cdots + a_k^g a_0^{h_k} \text{ and } a_i^f = a_1^g a_i^{h_1} + \cdots + a_k^g a_i^{h_k} \text{ for } 1 \leq i \leq n. \quad \Box
\]

**Lemma 2.4.3**  For every \( n \)-ary function \( f \in L_1 \), there exists a polynomial \( p_f \in \mathbb{N}[x_1, \ldots, x_n] \) such that \(|f(x_1, \ldots, x_n)| \leq p_f(|x_1|, \ldots, |x_n|)\) for all strings \( x_1, \ldots, x_n \).

**Proof**  The result is proved by induction on the definition of \( f \), where we use the notation \(|\overline{x}_n|\) to stand for the list \(|x_1|, \ldots, |x_n|\).

- If \( f \in L_0 \), then by the preceding lemma, \( p_f(|\overline{x}_n|) = a_0^f|x_1| + \cdots + a_n^f|x_n| \).
- If \( f \) is defined by CRN from \( h \), then as in the preceding lemma, \( p_f(|x|, |\overline{y}|) = |x| \).
- If \( f \) is defined by \( \text{COMP} \) from \( g \) and \( h_1, \ldots, h_k \), then

\[
|f(\overline{x})| = |g(h_1(\overline{x}), \ldots, h_k(\overline{x}))| \\
\leq p_g(|h_1(\overline{x})|, \ldots, |h_k(\overline{x})|) \\
\leq p_g(p_{h_1}(\overline{x}), \ldots, p_{h_k}(\overline{x}))
\]
(since polynomials in $\mathbb{N}[\vec{x}]$ are non-decreasing), so $p_f(|\vec{x}_n|) = p_g(p_{h_1}(|\vec{x}_n|), \ldots, p_{h_k}(|\vec{x}_n|))$.

- If $f$ is defined by TRN form $g, h, h_\ell, h_r$, where we assume without loss of generality that
  
  - $|g(x, z, \vec{y})| \leq p_g(|x|, |z|, |\vec{y}|)$,
  - $|h_\ell(z)| \leq c|z|$ and $|h_r(z)| \leq c|z|$,
  - $|h(x, z, \vec{y}, v_\ell, v_r)| \leq a_0 + a_1|x| + a_2|z| + \sum (b_i |y_i|) + a(|v_\ell| + |v_r|)$,

  then intuitively, at each level of the recursion, the length of the second argument is multiplied by $c$ so that at the bottom level (after $\lg |x|$ steps), the second argument has length $O(c^{|x||z|})$. At the same time, the lengths of each recursive call to $f$ are multiplied by $a$, which means that the total length of $f$ (bounded by the length of $g$ in the base case) is multiplied by $a^{\lg |x|}$. More precisely, we show that

  $$p_f(|x|, |z|, |\vec{y}|) = \left( a_0 + \sum (b_i |y_i|) \right) \cdot \sum_{j=0}^{\lg |x| - 1} (2a)^{j} + a_1 |x| \cdot \sum_{j=0}^{\lg |x| - 1} a^j$$

  $$+ a_2 |z| \cdot \sum_{j=0}^{\lg |x| - 1} (2ac)^{j} + (2a)^{\lg |x|} \cdot p_g(1, c^{\lg |x|} |z|, |\vec{y}|)$$

  $$\leq 2 |x|^{\lg a} + 1 \left[ a_0 + \sum (b_i |y_i|) + a_1 |x| + a_2 |z| |x|^{\lg c} 
  + a \cdot p_g(1, c |z| |x|^{\lg c}, |\vec{y}|) \right].$$

  Technically speaking, we need to use $\max\{1, |z|\}$ everywhere that $|z|$ appears in this expression, but this does not change the proof substantially besides making it longer to write down. Also, we need to deal separately with special cases such as when $a = 0$ or when the lengths of $h_\ell$ and $h_r$ are constants independent of $|z|$, but all of these cases simplify the proof so we present the general case only.

  Now, if $x = \varepsilon, 0, 1$, then $|f(x, z, \vec{y})| = |g(x, z, \vec{y})| \leq p_g(|x|, |z|, |\vec{y}|) \leq p_g(1, |z|, |\vec{y}|) \leq a_0 + a_1|x| + a_2|z| + \sum (b_i |y_i|) + p_g(1, |z|, |\vec{y}|) = p_f(1, |z|, |\vec{y}|)$. If $|x| > 1$, then we consider two subcases. If $|x|$ is even, then $\lfloor \lg(|x|/2) \rfloor = \lfloor \lg |x| \rfloor - 1$, so

  $$|f(x, z, \vec{y})| = |h(x, z, \vec{y}, f(x \triangledown, h_\ell(z), \vec{y}), f(\triangledown x, h_r(z), \vec{y}))|$$

  $$\leq a_0 + a_1|x| + a_2|z| + \sum (b_i |y_i|)$$

  $$+ a \left( |f(x \triangleleft, h_\ell(z), \vec{y})| + |f(\triangledown x, h_r(z), \vec{y})| \right)$$

  $\leq a_0 + a_1|x| + a_2|z| + \sum (b_i |y_i|)$$

  $$+ a(p_f(|x \triangledown|, |h_\ell(z)|, |\vec{y}|) + p_f(|\triangledown x|, |h_r(z)|, |\vec{y}|))$$

  $\leq a_0 + a_1|x| + a_2|z| + \sum (b_i |y_i|) + 2a p_f(|x|/2, c|z|, |\vec{y}|)$
\[ \begin{align*}
&\leq a_0 + a_1 |x| + a_2 |z| + \sum (b_i |y_i|) \\
&+ 2a \left[ \left( a_0 + \sum (b_i |y_i|) \right) \cdot \sum_{j=0}^{\lceil \lg |x| \rceil - 2} (2a)^j + a_1 \frac{|x|}{2} \cdot \sum_{j=0}^{\lceil \lg |x| \rceil - 2} a^j \\
&\quad + a_2 c |z| \cdot \sum_{j=0}^{\lceil \lg |x| \rceil - 2} (2ac)^j + (2a)^{\lceil \lg |x| \rceil - 1} \cdot p_y(1, c^{\lceil \lg |x| \rceil - 1} |z|, |\vec{y}|) \right] \\
&\leq p_f(|x|, |z|, |\vec{y}|).
\end{align*} \]

If \(|x|\) is odd, then \(\lceil \lg((|x| - 1)/2) \rceil = \lceil \lg |x| \rceil - 1\) and \(\lceil \lg((|x| + 1)/2) \rceil = \lceil \lg |x| \rceil - 1\), so

\[ |f(x, z, \vec{y})| = |h(x, z, \vec{y}, f(\langle \cdot x, h_{\ell}(z), \vec{y}, f(\langle \cdot x, h_r(z), \vec{y})\rangle)| \]

\[ \leq a_0 + a_1 |x| + a_2 |z| + \sum (b_i |y_i|) \\
+ a \left( |f(\langle \cdot x, h_{\ell}(z), \vec{y})| + |f(\langle \cdot x, h_r(z), \vec{y})| \right) \\
\leq a_0 + a_1 |x| + a_2 |z| + \sum (b_i |y_i|) \\
+ a (p_f(|\langle \cdot x|, |h_{\ell}(z)|, |\vec{y}|) + p_f(|\langle \cdot x|, |h_r(z)|, |\vec{y}|)) \\
\leq a_0 + a_1 |x| + a_2 |z| + \sum (b_i |y_i|) \\
+ a \left( a_0 + \sum (b_i |y_i|) \right) \cdot \sum_{j=0}^{\lceil \lg |x| \rceil - 2} (2a)^j + a_1 \frac{|x| - 1}{2} \cdot \sum_{j=0}^{\lceil \lg |x| \rceil - 2} a^j \\
\quad + a_2 c |z| \cdot \sum_{j=0}^{\lceil \lg |x| \rceil - 2} (2ac)^j + (2a)^{\lceil \lg |x| \rceil - 1} \cdot p_y(1, c^{\lceil \lg |x| \rceil - 1} |z|, |\vec{y}|) \\
+ a \left( a_0 + \sum (b_i |y_i|) \right) \cdot \sum_{j=0}^{\lceil \lg |x| \rceil - 2} (2a)^j + a_1 |x| \cdot \sum_{j=1}^{\lceil \lg |x| \rceil - 1} a^j \\
\quad + a_2 c |z| \cdot \sum_{j=1}^{\lceil \lg |x| \rceil - 1} (2ac)^j + (2a)^{\lceil \lg |x| \rceil - 1} \cdot p_y(1, c^{\lceil \lg |x| \rceil - 1} |z|, |\vec{y}|) \\
\quad + \frac{a_1}{2} \sum_{j=1}^{\lceil \lg |x| \rceil - 1} a^j + \frac{a_1}{2} \sum_{j=1}^{\lceil \lg |x| \rceil - 1} a^j \leq p_f(|x|, |z|, |\vec{y}|). \]
Now, we are ready to discuss circuits. For the sake of completeness, we summarize here Bloch’s definitions and results, suitably modified to apply to our setting. We will be working with circuit families that compute functions instead of relations, i.e., circuits will generally have multiple output gates. In this setting, we define $FNC^0$ to be the class of functions computed by constant depth circuit families (because the circuits have multiple output gates, this class contains interesting functions, unlike the relational counterpart $NC^0$), and $FNC^1$ to be the class of functions computed by logarithmic depth circuit families.

We assume that the gate set for our circuits consists of constants 0 and 1, unary identity ($\approx$) and negation ($\neg$), and binary conjunction ($\land$), disjunction ($\lor$), left projection ($\pi_L$), and right projection ($\pi_R$) (this could be reduced at the cost of longer proofs). Given a circuit family composed of such gates, we identify gates in the circuits by pairs $\langle \text{out}, \sigma \rangle$, where $\text{out}$ is the number of an output gate of the circuit (in binary) and $\sigma \in \{L, R, S\}^*$ represents a path in the circuit from the given output gate, where $L$ and $R$ indicate the left and right inputs of a binary gate, respectively, and $S$ indicates the only input of a unary gate. (Note that we number the output gates from right to left, starting with 1, and similarly for the input gates of each input.)

Now, we want to work with uniform families of circuits. Unfortunately, the standard notion of $U_E$-uniformity defined by Ruzzo [28] does not make sense for constant depth circuits (it would require the extended connection language for the circuits to be recognizable by an ATM in constant time, which is not even enough time for the ATM to examine a gate number or an input length). To remedy this, Bloch defines a notion of mapping-uniform circuits, where the uniformity computation is divided in two phases. The main purpose of the uniformity computation is to be able to recognize connections in the circuit (i.e., given a gate and a path, what gate is at the end of the path?) and gate information (i.e., given a gate number, what type is that gate?). Because of our numbering scheme for gates, determining the descendant of a gate $\langle \text{out}, \sigma \rangle$ along a path $\sigma'$ is easy: the answer is simply $\langle \text{out}, \sigma \sigma' \rangle$. Determining the type of an internal gate is also not difficult, as we will see. The hard part, requiring the “two-phase” approach, is to determine which input bit is tied to an input gate (which is necessary to fully specify the “type” of that input gate).

Here is some intuition behind the two phases of the uniformity computation. Essentially, given an input gate $\langle \text{out}, \sigma \rangle$ in a circuit, the first phase must use $\sigma$ to identify how to compute the number of the input bit as a function of the output gate number, but without carrying out that computation. This will be done by a deterministic Turing machine running in time proportional to $|\sigma|$, which is bounded by the depth of the circuit, and it is this phase of the computation that will be composed or iterated for functions defined by composition or recursion. The second phase of the computation will only be carried out once, by an ATM that combines the information from the first phase together with the rest of the information about the circuit.
More precisely, the first phase will output a term $t_\sigma$ that may be thought of as mapping output bit positions to input bit positions that “affect” that output bit, i.e., given an input gate $\langle \text{OUT}, \sigma \rangle$ in the circuit tied to bit number $r$ of some input parameter, we want $t_\sigma(\text{OUT}) = r$.

For this purpose, we introduce the mapping language of a family of circuits, which consists of a set of functions that encapsulate all the “primitive” dependencies that may exist between output and input bit numbers. Besides the natural numbers used to represent bit positions, we also use the symbol “⊥” to indicate that a given output bit does not depend on any input bit.

For a circuit with $k$ input parameters $y_1, \ldots, y_k$, the mapping language contains the following function symbols (each function is implicitly defined to be equal to ⊥ when its argument is ⊥).

- $\text{UNDEF}(x) = ⊥$
- $\text{ONE}(x) = 1$
- $\text{ADD}_j(x) = x + |y_j|$
- $\text{MIN}_j(x) = \min\{x, |y_j|\}$
- $\text{SUB}_j(x) = x - |y_j|$

Before we move back to circuits, we argue that terms $t_\sigma$ in the mapping language can be computed in alternating logarithmic time (as a function of $m = \max\{|t_\sigma|, \text{OUT}, |y_1|, \ldots, |y_k|\}$).

Given $t_\sigma, \text{OUT}, |y_1|, \ldots, |y_k|$, an ATM can check in parallel if $t_\sigma$ contains UNDEF and output ⊥ immediately if this is the case; otherwise, the ATM guesses the position in $t_\sigma$ where the last “ONE” appears, and replaces the subsequent part of the term with 1. Once these simple checks are done, the ATM can construct lists of numbers from $|y_1|, \ldots, |y_k|$ and subterms of $t_\sigma$ for each block of functions of the form $\text{MIN}_{j_1}(\ldots \text{MIN}_{j_\ell}(t') \ldots )$ in $t_\sigma$, and evaluate each of these blocks in parallel. The rest of the subterms will contain only $\text{ADD}_j$ and $\text{SUB}_j$ functions, and these lists of terms and numbers from $|y_1|, \ldots, |y_k|$ can be added and subtracted (using two’s complement) with standard carry-save techniques.

Now, following Bloch, we say that a circuit family is mapping-uniform if there exist a deterministic multi-tape Turing machine $P$ and an ATM $Q$ such that for every gate $\langle \text{OUT}, \sigma \rangle$ in a circuit of the family, the following two conditions hold.

1. If $\langle \text{OUT}, \sigma \rangle$ is an input gate tied to bit number $r$ of some input parameter, then $P$ on input $\sigma$ runs in time $O(|\sigma|)$ and outputs a term $t_\sigma$ in the mapping language such that $t_\sigma(\text{OUT}) = r$.

2. Machine $Q$ on input $\text{OUT}, \sigma, |y_1|, \ldots, |y_k|, \tau$ runs in logarithmic time, i.e., in alternating time $O(\log(\max\{|\text{OUT}, |\sigma|, |y_1|, \ldots, |y_k|\}))$ and accepts iff $\langle \text{OUT}, \sigma \rangle$ is an internal gate of type $\tau$ or $\langle \text{OUT}, \sigma \rangle$ is an input gate tied to bit number $r$ of $y_j$ and $\tau = \langle j, r \rangle$ (in some standard encoding).

A direct argument shows that any mapping-uniform family of circuits of at least logarithmic depth is also $U_{E^*}$-uniform. Now, we are ready to show that $L_1 \subseteq FNC^1$. 
Chapter 2. The String Algebra $L_1$

Lemma 2.4.4  For every $f \in \text{BASE}$, $f$ can be computed by a uniform circuit family in $\text{FNC}^0$.

Proof

$\varepsilon$: The empty circuit is the only one computing this function. Hence, on input $\sigma = \varepsilon$, $P$ outputs $\text{UNDEF}(x)$; on input $\text{OUT}, \sigma, \tau$, $Q$ accepts iff $\text{OUT} = 0$, $\sigma = \varepsilon$, and $\tau = \varepsilon$.

$0, 1$: Circuits for these functions consist of a single output gate, one of the “constant” gates 0 or 1, appropriately. Hence, on input $\sigma = \varepsilon$, $P$ outputs $\text{UNDEF}(x)$; on input $\text{OUT}, \sigma, \tau$, $Q$ accepts iff $\text{OUT} = 1$, $\sigma = \varepsilon$, and $\tau = 0$ or $\tau = 1$, respectively.

$\uparrow y_1$: Circuits for this function connect output gates to input gates directly, using unary identity gates $\approx$. Hence, on input $\sigma = S$, $P$ outputs simply $x$; on input $\text{OUT}, \sigma, |y_1|, \tau$, $Q$ accepts iff $1 \leq \text{OUT} \leq \lfloor |y_1|/2 \rfloor$, $\sigma = \varepsilon$ and $\tau = \approx$, or $\sigma = S$ and $\tau = \langle 1, \text{OUT} \rangle$.

$y_1 \cdot y_2$: Again, circuits for this function connect output gates to input gates directly, using unary identity gates $\approx$. Hence, on input $\sigma = S$, $P$ outputs $\text{ADD}_2(x)$; on input $\text{OUT}, \sigma, |y_1|, |y_2|, \tau$, $Q$ accepts iff $1 \leq \text{OUT} \leq |y_1| - |y_2|$, $\sigma = \varepsilon$ and $\tau = \approx$, or $\sigma = S$ and $\tau = \langle 1, r \rangle$ for $r = \text{ADD}_2(\text{OUT})$.

$y_1 \cdot y_2$: The simplest circuits to compute this function would use unary identity gates connected directly to the input bits, as in the last two cases. Unfortunately, this would not allow $P$ to know from $\sigma$ alone which term to output. Therefore, we do something slightly different, as depicted in Figure 2.4.1.

![Figure 2.4.1: Uniform circuits for the concatenation function ",".](image)

Now, on input $\sigma = L$, $P$ outputs $\text{SUB}_2(x)$, while on input $\sigma = R$, $P$ outputs simply $x$; on input $\text{OUT}, \sigma, |y_1|, |y_2|, \tau$, $Q$ accepts iff

- $1 \leq \text{OUT} \leq |y_2|$ and
  - $\sigma = \varepsilon$, $\tau = \pi_R$, or
  - $\sigma = L$, $\tau = 0$, or
  - $\sigma = R$, $\tau = \langle 2, \text{OUT} \rangle$; or
- $|y_2| + 1 \leq \text{OUT} \leq |y_2| + |y_1|$ and
  - $\sigma = \varepsilon$, $\tau = \pi_L$, or
  - $\sigma = R$, $\tau = 0$, or
  - $\sigma = L$, $\tau = \langle 1, r \rangle$ for $r = \text{SUB}_2(\text{OUT})$.  


2.4. \textit{L}_1 \text{ AND} \textit{FALOGTIME}

\textbf{\textit{y}_1 \ ? (\textit{y}_2, \textit{y}_3, \textit{y}_4)}: This is the only base function requiring circuits of depth greater than one. There is one consideration making the circuits slightly more complicated than it would seem necessary at first: the shorter of the last two input parameters must be “padded” to the same length as the longer, requiring some extra gates. So, the circuits for \textit{?} are of two different kinds: when $|y_1| = 0$, the circuits simply use unary identity gates $\approx$ for output, connected directly to the input gates of $y_2$, while if $|y_1| > 0$, the circuits are depicted in Figure 2.4.2 (we illustrate the case when $0 < |y_3| < |y_4|$; the other cases are identical except for the obvious modifications to the types of the projection gates).

Figure 2.4.2: Uniform circuits for the conditional function “?”.

Now, there are only a constant number of possibilities for $\sigma$ that machine $P$ needs to check. We list each one and the corresponding output for $P$, as well as a brief explanation indicating which bit of which input parameter is designated by the given path $\sigma$, in Table 2.4.1.

| $\sigma$ | output | explanation |
|---------|--------|-------------|
| $S$     | $x$    | same bit of $y_2$ |
| $RL$    | $\text{ONE}(x)$ | first bit of $y_1$ |
| $LLS$   | $\text{ONE}(x)$ | first bit of $y_1$ (negated) |
| $LRR$   | $\text{MIN}_3(x)$ | bit of $y_3$ (padded) |
| $RRR$   | $\text{MIN}_4(x)$ | bit of $y_4$ (padded) |

Table 2.4.1: Behaviour of machine $P$ for the conditional function.

Next, on input $\text{OUT}, \sigma, |y_1|, |y_2|, |y_3|, |y_4|, \tau$, $Q$ accepts iff

- $|y_1| = 0, 1 \leq \text{OUT} \leq |y_2|$, and
  - $\sigma = \varepsilon, \tau = \approx$, or
  - $\sigma = S, \tau = \langle 2, \text{OUT} \rangle$; or
• \(|y_1| > 0, 1 \leq \text{out} \leq \max\{|y_3|, |y_4|\},\) and

- \(\sigma = \varepsilon, \tau = \lor,\) or
- \(\sigma = L, \tau = \land,\) or
- \(\sigma = R, \tau = \land,\) or
- \(\sigma = LL, \tau = \lnot,\) or
- \(\sigma = LR, \tau = \pi_R\) if \(\text{out} \leq |y_3|, \tau = \pi_L\) if \(|y_3| < \text{out},\) or
- \(\sigma = RL, \tau = \langle 1,1 \rangle,\) or
- \(\sigma = RR, \tau = \pi_R\) if \(\text{out} \leq |y_4|, \tau = \pi_L\) if \(|y_4| < \text{out},\) or
- \(\sigma = LLS, \tau = \langle 1,1 \rangle,\) or
- \(\sigma = LRL, \tau = 0,\) or
- \(\sigma = LRR, \tau = \langle 3,r \rangle\) for \(r = \text{min}_3(\text{out}),\) or
- \(\sigma = RRL, \tau = 0,\) or
- \(\sigma = RRR, \tau = \langle 4,r \rangle\) for \(r = \text{min}_4(\text{out}).\)

\(T_k^n(y_1, \ldots, y_n):\) Circuits will use unary identity gates \(\approx\) directly connected to the proper input bits. Hence, on input \(\sigma = S, P\) outputs simply \(x;\) on input \(\text{out}, \sigma, |y_1|, \ldots, |y_n|, \tau, Q\) accepts iff \(1 \leq \text{out} \leq |y_k|, \sigma = \varepsilon\) and \(\tau = \approx,\) or \(\sigma = S\) and \(\tau = \langle k, \text{out} \rangle.\)

Next, we want to show that functions defined by CRN also have uniform circuit families. For technical reasons (i.e., to simplify the proof), we actually show the result for left CRN. (Since \(L_0\) and \(L_1\) remain the same whether CRN or left CRN is used to define them, this is sufficient.)

**Lemma 2.4.5** If \(f(y_1, y_2, \ldots, y_n)\) is defined from \(h\) by left CRN on \(y_1,\) where \(h\) has uniform circuits of depth \(d_h(|y_1|, |y_2|, \ldots, |y_n|),\) then \(f\) has uniform circuits of depth

\(\max\{d_h(1, |y_2|, \ldots, |y_n|), d_h(2, |y_2|, \ldots, |y_n|), \ldots, d_h(|y_1|, |y_2|, \ldots, |y_n|)\}.\)

**Proof** A natural circuit for \(f\) consists of a series of \(h\)-circuits in parallel, one for each output bit of \(f,\) where the \(i\)-th \(h\)-circuit is connected to the first \(i\) bits of \(y_1\) (as well as to every other input parameter). Clearly, the depth of this circuit is as stated above. Moreover, given a path \(\sigma,\) machine \(P\) simply simulates \(P_h\) to get a term \(t'_{\sigma}(x) = t_{\sigma}^h(\text{one}(x)),\) while machine \(Q\) accepts \(\text{out}, \sigma, |y_1|, |y_2|, \ldots, |y_n|, \tau\) iff \(|y_1| = 0, \text{out} = 0, \sigma = \varepsilon, \tau = \varepsilon,\) or \(Q_h\) accepts \(\text{out}' = 1, \sigma, |y_1'| = \text{out}, |y_2|, \ldots, |y_n|, \tau.\)

Following this, we need to show that the composition of functions computed by uniform families of circuits is also computable by uniform families of circuits, of the right depth.
Lemma 2.4.6  If $f(y_1, \ldots, y_n)$ is defined from $g$ and $h_1, \ldots, h_k$ by COMP, where $g$ has uniform circuits of depth $d_g(|z_1|, \ldots, |z_k|)$ and $h_i$ has uniform circuits of depth $d_{h_i}(|y_1|, \ldots, |y_n|)$ (for $1 \leq i \leq k$), then $f$ has uniform circuits of depth

$$d_g(|h_1(\vec{y}_n)|, \ldots, |h_k(\vec{y}_n)|) + \max\{d_{h_i}(|\vec{y}_n|)\} + \lceil \lg(k) \rceil.$$  

Proof A natural circuit for $f$ consists of a circuit for $g$ whose input gates are connected to the output gates of the corresponding circuits for $h_i$ directly. Unfortunately, machine $P$ cannot tell what term to output just from a path in such a circuit, because $P$ does not have enough time to determine which $h_i$ the path leads into (this would require determining the number of the input gate of $g$ through which the path passes). Therefore, we construct a slightly more complicated circuit by adding a layer of selection gates of depth $\lceil \lg(k) \rceil$ between the circuit for $g$ and the $h_i$ circuits (somewhat like what was done for the concatenation function), in such a way that machine $P$ can easily determine from a path $\sigma$ which $h_i$ the path goes through. Clearly, the depth of such a circuit is as stated above. Moreover, given a path $\sigma$, machine $P$ can break it up into $\sigma_g$ through $g$ (getting a term $t_{\sigma_g}$), followed by $\sigma'$ through the selection subcircuit (which gives the index $i$ of the function $h_i$ feeding into the $g$ circuit), followed by a final part $\sigma_i$ through $h_i$ (getting a term $t_{\sigma_i}$). $P$ then outputs $t_{\sigma_i}(t_{\sigma_g}(x))$, in time linear in $|\sigma|$. Also, on input output, $\sigma, |y_1|, \ldots, |y_n|, \tau$, machine $Q$ accepts iff $\langle \text{output}, \sigma \rangle$ is a gate of type $\tau$ in $g$ (by simulating $Q_g$), or $\langle \text{output}, \sigma \rangle$ is a gate within $\lceil \lg(k) \rceil$ steps of an input of $g$ and $\tau$ is the correct type of projection gate (computing $t_{\sigma_g}(\text{output})$ to figure out which input bit of $g$ the path $\sigma$ goes through, and then checking the constantly many possibilities for the part of $\sigma$ through the selection subcircuit), or $\langle \text{output}, \sigma \rangle$ is a gate of type $\tau$ or an input gate $\tau = \langle j, r \rangle$ for $r = t_{\sigma_i}(t_{\sigma_g}(\text{output}))$ within an $h_i$ circuit (computing $t_{\sigma_g}(\text{output})$ and tracking the path through the selection subcircuit to figure out the index $i$, and then simulating $Q_{h_i}$ to verify $\tau$). All this can be done in logarithmic time. \hfill \Box

Finally, we show that functions defined by TRN can be computed by uniform families of circuits.

Lemma 2.4.7  If $f(y_1, y_2, y_3, \ldots, y_n)$ is defined from $g$, $h$, $h_\ell$, and $h_r$ by TRN, where $g$ has uniform circuits of depth $d_g(|y_1|, |y_2|, |y_3|, \ldots, |y_n|)$, $h$ has uniform circuits of depth $d_h$ (a constant), and $h_\ell$ and $h_r$ have uniform circuits of depth $d_\ell$ and $d_r$ (constants), then $f$ has uniform circuits of depth

$$O(d_g(1, |y_2||y_1|^c, |y_3|, \ldots, |y_n|) + \log |y_1| \cdot (d_h + d_\ell + d_r))$$

for some constant $c \in \mathbb{N}$. 
PROOF A natural circuit for $f$ consists of a binary tree of $h$-subcircuits connected to the appropriate bits of the first input and to each other, with a layer of $g$ circuits at the bottom, where the second input of the $h$ and $g$ circuits is connected to subtrees of $h_\ell$ and $h_r$ circuits. As in the proof for composition, we use layers of selection gates to “glue” together the different subcircuits (between successive $h$ circuits, between $h$ and $g$ circuits, between $h$ or $g$ circuits and the circuits to compute left and right halves of the first input or left and right functions of the second input, as well as between successive “half” functions for the first and second inputs), so that $P$ can tell from the path $\sigma$ alone which subcircuit a path leads to. The total depth of such a circuit is obviously as stated. Moreover, given a path $\sigma$, $P$ can divide the path into portions through $h$ circuits, accumulating the terms for each one, and the final portion of the path through an $h$, $g$, $h_\ell$, $h_r$, or “half” circuit, outputting the composition of each term. This can obviously be done in linear time in the length of $\sigma$. On input $\text{OUT}, \sigma, |y_1|, |y_2|, |y_3|, \ldots, |y_n|, \tau$, machine $Q$ can break up the path $\sigma$ into a first part through some number of $h$ circuits and a final part $\sigma'$ entirely contained inside some $h$, $g$, $h_\ell$, $h_r$, or “half” subcircuit. $Q$ can compute the term corresponding to the first part of the path to figure out which output bit of the subcircuit $\sigma$ passes through, and then simulate the ATM for that subcircuit to verify $\tau$, all in logarithmic time. □

Now, we can put all of these results together.

**Theorem 2.4.1** For all $f \in L_0$, $f$ can be computed by a uniform family of circuits in $FNC^0$ (i.e., of constant depth).

**Proof** By induction on the definition of $f$: Lemma 2.4.4 shows that functions in $\text{BASE}$ have uniform constant depth circuits, and Lemmas 2.4.5 and 2.4.6 show that CRN and COMP preserve uniform constant depth. □

**Theorem 2.4.2** For all $f \in L_1$, $f$ can be computed by a uniform family of circuits in $FNC^1$ (i.e., of logarithmic depth).

**Proof** By induction on the definition of $f$.

- If $f \in L_0$, then the preceding theorem shows the result.
- If $f$ is defined by CRN from $h \in L_1$, then Lemma 2.4.5 shows that $f$ can be computed by uniform circuits of the same depth as that of the circuits for $h$, which shows the result.
- If $f$ is defined by COMP from $g, h_1, \ldots, h_k \in L_1$, then Lemma 2.4.6 and Lemma 2.4.3 (on
the length of functions in \( L_1 \) show that \( f \) can be computed by uniform circuits of depth

\[
d_g(|h_1(\vec{y})|, \ldots, |h_k(\vec{y})|) + \max\{d_{h_1}(|\vec{y}|), \ldots, d_{h_k}(|\vec{y}|)\} + \lfloor \log(k) \rfloor
\]

\[
= \mathcal{O}\left( \log \left( \max\{p_{h_1}(|\vec{y}|), \ldots, p_{h_k}(|\vec{y}|)\} + \log(\max\{|\vec{y}|\}) \right) \right)
\]

\[
= \mathcal{O}(\log(\max\{|\vec{y}|\})).
\]

• If \( f \) is defined by TRN from \( g \in L_1 \) and \( h, h_\ell, h_r \in L_0 \), then Lemma 2.4.7, together with Lemmas 2.4.2 and 2.4.3 on the length of functions in \( L_0 \) and \( L_1 \), shows that \( f \) can be computed by uniform circuits of depth

\[
\mathcal{O}(d_g(1, |y_2|, |y_3|, \ldots, |y_n|) + \log |y_1| \cdot (d_h + d_\ell + d_r))
\]

\[
= \mathcal{O}(\log(\max\{|y_2|, \ldots, |y_n|\}) + \log |y_1|)
\]

\[
= \mathcal{O}(\log(\max\{|y_1|, |y_2|, \ldots, |y_n|\})). \quad \square
\]
Chapter 3

The Quantifier-Free Theory $T_1$

In this chapter, we will define the theory $T_1$ and give its formal development, including many proofs of simple properties of functions of $T_1$, as well as many derived rules, and concluding with an illustrative example by proving the pigeonhole principle.

3.1 Definitions

The theory $T_1$ that we now describe is a quantifier-free system, i.e., a free-variable theory with propositional connectives, modeled after Cook’s $PV$ \cite{18} but based on the algebra $L_1$. The language of $T_1$ consists of the function symbols

$$\{\varepsilon, 0, 1, 0\cdot, 1\cdot, \triangleleft, \triangleright, \cdot, \triangleleft, \triangleleft, \?\},$$

the function constructors $\{\lambda, \ell CRN, r CRN, TRN\}$, the predicate symbol $\{=\}$, and the usual propositional connectives $\{\neg, \&, \vee, \rightarrow, \leftrightarrow\}$. More precisely, we have the following definitions (where we use the informal notation $x0$ for $(x \cdot 0)$ and $0x$ for $(0 \cdot x)$—similarly for $x1$ and $1x$).

**Definition 3.1.1** The function symbols and terms of $T_1$ are defined as follows. (The intended interpretation of each function symbol is as given in Chapter 2 where $\ell CRN$ represents “left” (or “reverse”) CRN and $r CRN$ represents “right” (or “plain”) CRN, and we use the notation introduced there instead of the more formal prefix notation. Also, each function symbol and each term has a rank of either 0 or 1—that intuitively indicates which one of $L_0$ or $L_1$ the function symbol or term belongs to.)

1. Each variable $x_0, x_1, x_2, \ldots$ is a term of rank 0.

2. If $f$ is an $n$-place function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term whose rank is the maximum of the ranks of $f, t_1, \ldots, t_n$ (i.e., the rank of $f(t_1, \ldots, t_n)$ is 0 iff the rank of each one of $f, t_1, \ldots, t_n$ is 0).
3. \( \varepsilon, 0, 1 \) are 0-place function symbols (constants) of rank 0.

4. \( 0, 1, \blacktriangleleft, \blacktriangleright \) are 1-place function symbols of rank 0.

5. \( \cdot, \blacktriangleleft, \blacktriangleright \) are 2-place function symbols of rank 0.

6. \(?\) is a 3-place function symbol of rank 0.

7. If \( t \) is a term and \( x_1, \ldots, x_n \) is a list of variables including all the variables in \( t \), then \([\lambda x_1 \ldots x_n.t] \) is an \( n \)-place function symbol of the same rank as that of \( t \).

8. If \( h \) is an \((n + 1)\)-place function symbol, then \( \ell \text{CRN}[h] \) and \( r \text{CRN}[h] \) are \((n + 1)\)-place function symbols whose rank is that of \( h \).

9. If \( g \) is an \((n + 2)\)-place function symbol, \( h \) is an \((n + 4)\)-place function symbol of rank 0, and \( h_\ell \) and \( h_r \) are 1-place function symbols of rank 0, then \( \text{TRN}[g, h, h_\ell, h_r] \) is an \((n + 2)\)-place function symbol of rank 1.

**Definition 3.1.2** The axioms of \( T_1 \) are as follows (except for the propositional and equality axioms, they simply define the function symbols).

0. Any standard, complete set of axioms for the propositional calculus (with equations of the form \( x = y \) in place of propositional atoms, for arbitrary variables \( x \) and \( y \)).

1. (a) \( x = x \)
   (b) \( x = y \rightarrow y = x \)
   (c) \( (x = y \land y = z) \rightarrow x = z \)
   (d) \( (x_1 = y_1 \land \cdots \land x_k = y_k) \rightarrow f(x_1, \ldots, x_k) = f(y_1, \ldots, y_k) \)
   (for all \( k \)-ary function symbols \( f \), for all \( k \geq 1 \))

2. \( \varepsilon \neq 0 \land 0 \neq 1 \land 1 \neq \varepsilon \)

3. (a) \( x \cdot \varepsilon = x \land x \cdot 0 = (x \cdot y) \cdot 0 \land x \cdot y 1 = (x \cdot y) \cdot 1 \)
   (b) \( x \cdot y = \varepsilon \rightarrow (x = \varepsilon \land y = \varepsilon) \)
   (c) \( x \cdot y = 0 \rightarrow (x = \varepsilon \land y = 0) \lor (x = 0 \land y = \varepsilon) \)
   \( x \cdot y = 1 \rightarrow (x = \varepsilon \land y = 1) \lor (x = 1 \land y = \varepsilon) \)

4. (a) \( \varepsilon \blacktriangleright x = x \land 0 \blacktriangleright x = 0 \blacktriangleright (y \blacktriangleright x) \land 1 \blacktriangleright x = 1 \blacktriangleright (y \blacktriangleright x) \)
   (b) \( 0 \blacktriangleright \varepsilon = \varepsilon \land 0 \blacktriangleright 0x = x \land 0 \blacktriangleright 1x = x \land 1 \blacktriangleright x = 1 \blacktriangleright 1x = x \)
   \( 1 \blacktriangleright \varepsilon = \varepsilon \land 1 \blacktriangleright 0x = x \land 1 \blacktriangleright 1x = x \)
   (c) \( y \blacktriangleright x = \varepsilon \leftrightarrow x \blacktriangleright y 0 \neq \varepsilon \leftrightarrow x \blacktriangleright y 1 \neq \varepsilon \)
5. (a) \( x = x \triangleleft \varepsilon \land (x \triangleleft y) \triangleleft 0 = x \triangleleft y0 \land (x \triangleleft y) \triangleleft 1 = x \triangleright y1 \)
   (b) \( \varepsilon = \varepsilon \triangleleft 0 \land x = x0 \triangleleft 0 \land x = x1 \triangleleft 0 \)
   \( \varepsilon = \varepsilon \triangleleft 1 \land x = x0 \triangleleft 1 \land x = x1 \triangleleft 1 \)
   (c) \( \varepsilon \neq 1y \triangleleft x \iff \varepsilon \neq 0y \triangleleft x \iff \varepsilon = x \triangleleft y \)
6. (a) \( 0\varepsilon = \varepsilon \land 0(x0) = 0x \cdot 0 \land 0(x1) = 0x \cdot 0 \)
   (b) \( 1\varepsilon = \varepsilon \land 1(x0) = 1x \cdot 1 \land 1(x1) = 1x \cdot 1 \)
7. \( \varepsilon ? (x, y, z) = x \land w0 ? (x, y, z) = 0(z \triangleleft y) \cdot y \land w1 ? (x, y, z) = 0(y \triangleleft z) \cdot z \)
8. (a) \( (x \triangleleft \|x\rangle) \cdot (\|x\rangle x) = x \)
   (b) \( (x \triangleleft \|x\rangle \triangleright x) = \varepsilon \land 1 \triangleright (x \triangleleft \|x\rangle \triangleright x) = \varepsilon \)
9. We use “\( x ?_{EL} (y, z) \)” as a shorthand notation for \( (x \triangleleft \|x\rangle \triangleright x) \cdot (y, z, z) \) (which equals \( y \) if the length of \( x \) is even, \( z \) if the length of \( x \) is odd).
   (a) \( (x0)\triangleleft = x ?_{EL} (x \triangleleft, x \triangleleft \cdot ((\|x\rangle \cdot 0) \triangleleft \|x\rangle)) \land (x1)\triangleleft = x ?_{EL} (x \triangleleft, x \triangleleft \cdot ((\|x\rangle \cdot 1) \triangleleft \|x\rangle)) \)
   (b) \( \|x0\rangle = x ?_{EL} (\|x\rangle \cdot 0, 1 \triangleright (\|x\rangle \cdot 0)) \land \|x1\rangle = x ?_{EL} (\|x\rangle \cdot 1, 1 \triangleright (\|x\rangle \cdot 1)) \)
   (c) \( (0x)\triangleleft = x ?_{EL} ((0 \cdot x \triangleleft) \triangleleft 1, 0 \cdot x \triangleleft) \land (1x)\triangleleft = x ?_{EL} ((1 \cdot x \triangleleft) \triangleleft 1, 1 \cdot x \triangleleft) \)
   (d) \( \|0x\rangle = x ?_{EL} ((x \triangleleft \cdot (0 \cdot x \triangleleft)) \cdot \|x\rangle, \|x\rangle) \land \|1x\rangle = x ?_{EL} ((x \triangleleft \cdot (1 \cdot x \triangleleft)) \cdot \|x\rangle, \|x\rangle) \)
10. \( [\lambda x_1 \ldots x_n \cdot t](x_1, \ldots, x_n) = t \)
11. (a) \( \ell \text{CRN}[h](\varepsilon, \vec{y}) = \varepsilon \land \ell \text{CRN}[h](0x, \vec{y}) = ((h(0x, \vec{y}) \cdot 0) \triangleleft h(0x, \vec{y})) \cdot \ell \text{CRN}[h](x, \vec{y}) \)
   (b) \( r \text{CRN}[h](\varepsilon, \vec{y}) = \varepsilon \land r \text{CRN}[h](x0, \vec{y}) = r \text{CRN}[h](x, \vec{y}) \cdot (h(x0, \vec{y}) \triangleright (0 \cdot h(x0, \vec{y}))) \)
12. \( \text{TRN}[g, h, h_\ell, h_r](x, z, \vec{y}) = x \triangleleft 1 ? (g(x, z, \vec{y}), t, t) \)

Remark 3.1.1 By Claim 2.4.1 every function in \textsc{FALOGTIME} is represented by some function symbol in \( T_1 \), and every function symbol in \( T_1 \) represents a function in \textsc{FALOGTIME}.

Definition 3.1.3 The rules of inference of \( T_1 \) are as follows.

0. Any standard, complete set of rules for the propositional calculus.
1. **Substitution** for an arbitrary formula $A$, variable $x$, and term $t$:
   \[ A \vDash A[t/x] \]

2. **Induction on Notation** (NIND) for an arbitrary formula $A$ and variable $x$:
   (a) ("left" version) $A[\varepsilon], A[x] \rightarrow A[0x], A[x] \rightarrow A[1x] \vDash A$
   (b) ("right" version) $A[\varepsilon], A[x] \rightarrow A[x0], A[x] \rightarrow A[x1] \vDash A$

3. **Tree Induction** (TIND) for an arbitrary formula $A$, variables $x, z$, and unary function symbols $h_\ell, h_r$ of rank 0:
   \[ A[\varepsilon, z], A[0, z], A[1, z], (A[x \triangleleft h_\ell(z)] \land A[\triangleright x, h_r(z)]) \rightarrow A[x, z] \vDash A \]

### 3.2 Developing the theory

In this section, we give a formal development of $T_1$, starting with a few simple theorems and working our way towards multi-variable versions of CRN and NIND. These will be used to define binary addition and "counting" functions, and to prove their properties.

#### 3.2.1 Basic definitions and theorems

**Claim 3.2.1** $\varepsilon \cdot x = x$

**Proof** By NIND on $x$: $\varepsilon \cdot \varepsilon = \varepsilon$ (by Axiom [3a] and Rule [1]),
\[
\begin{align*}
\varepsilon \cdot x0 &= (\varepsilon \cdot x) \cdot 0 & (\text{Axiom [3a] Rule [1]}), \\
&= x \cdot 0 & (\text{Induction Hypothesis, Axiom [1d] Rule [1]}), \\
\varepsilon \cdot x1 &= (\varepsilon \cdot x) \cdot 1 & (\text{Axiom [3a] Rule [1]}), \\
&= x \cdot 1 & (\text{Induction Hypothesis, Axiom [1d] Rule [1]}). \quad \Box
\end{align*}
\]

**Claim 3.2.2** $x \cdot (y \cdot z) = (x \cdot y) \cdot z$

**Proof** By NIND on $z$: $x \cdot (y \cdot \varepsilon) = x \cdot y = (x \cdot y) \cdot \varepsilon$ (by Axiom [3a] and Rule [1]),
\[
\begin{align*}
x \cdot (y \cdot z0) &= x \cdot ((y \cdot z) \cdot 0) & (\text{Axioms [3a] and [1d] Rule [1]}), \\
&= (x \cdot (y \cdot z)) \cdot 0 & (\text{Axioms [3a] and Rule [1]}), \\
&= ((x \cdot y) \cdot z) \cdot 0 & (\text{Induction Hypothesis}), \\
&= (x \cdot y) \cdot z0 & (\text{Axioms [3a] and Rule [1]}), \\
\end{align*}
\]
and a similar proof shows $x \cdot (y \cdot z1) = (x \cdot y) \cdot z1. \quad \Box$
3.2. Developing the theory

Note that in the proofs that follow, we will not mention explicitly the application of particular axioms, of the induction hypothesis, or of the substitution rule when they are self-evident. Also, when proving a statement by NIND, the cases for \( x_0 \) and \( x_1 \) will often be almost identical (as above) so we will prove both cases at once using “\( i \)” to stand for either 0 or 1.

**Remark 3.2.1** Be advised that the rest of this section contains a large number of technical claims, together with their proofs, which are included here for the sake of completeness. Most of these claims are of limited interest in themselves, apart from illustrating the style of proofs in \( T_1 \) and giving basic properties of functions which will be used in later proofs. For this reason, we recommend that on a first reading, the reader focus mainly on the Definitions, Theorems, and Derived Rules, which contain the essential results.

**On “\( \triangleright \)” and “\( \triangleleft \)”**

We start by defining two functions that will serve as a convenient shorthand notation throughout the rest of this chapter, and prove basic properties of these functions.

**Definition 3.2.1**

\[
\begin{align*}
(\text{L}) \; \triangleright & = [\lambda x. \; 1 \triangleright x] \\
(\text{R}) \; \triangleleft & = [\lambda x. \; x \triangleleft 1]
\end{align*}
\]

(To make the notation more consistent with previous usage, we will write “\( \triangleright x \)” and “\( x \triangleleft \)” instead of the more formal “\( \triangleright(x) \)” and “\( x(\triangleleft) \)” respectively.)

**Claim 3.2.3**

\[
\begin{align*}
(\text{L}) \; 0 \triangleright x & = 1 \triangleright x \\
(\text{R}) \; x \triangleleft 1 & = x \triangleleft 0
\end{align*}
\]

**Proof** (L): Immediate from Axiom 4b. (R): Immediate from Axiom 5b. \( \square \)

**Claim 3.2.4**

\[
\begin{align*}
(\text{L}) \; \triangleright \varepsilon & = \varepsilon \land \triangleright (0x) = x \land \triangleright (1x) = x \\
(\text{R}) \; \varepsilon \triangleleft & = \varepsilon \land (x0) \triangleleft = x \land (x1) \triangleleft = x
\end{align*}
\]

**Proof** (L): This is just a restatement of Axiom 4b. (R): This is just a restatement of Axiom 5b. \( \square \)

**Claim 3.2.5**

\[
\begin{align*}
(\text{L}) \; y \triangleright \varepsilon & = \varepsilon \\
(\text{R}) \; \varepsilon & = \varepsilon \triangleleft y
\end{align*}
\]

**Proof** (L) By NIND on \( y \): \( \varepsilon \triangleright \varepsilon = \varepsilon \), \( iy \triangleright \varepsilon = i \triangleright (y \triangleright \varepsilon) = i \triangleright \varepsilon = \varepsilon \). (R) By NIND on \( y \):

\[
\varepsilon \triangleright \varepsilon = \varepsilon, \varepsilon \triangleleft y i = (\varepsilon \triangleleft y) \triangleleft i = \varepsilon \triangleleft i = \varepsilon.
\]

Many of the theorems that follow will be similar to the ones above in having a “left” and “right” version, both of which can be proved in the same way (by using the appropriate version of NIND when necessary). Hence, to avoid unnecessary repetition, we will only give the proof of one version from now on.
Claim 3.2.6  \( (L) \Rightarrow(y \triangleright x) = y \triangleright \triangleright x \quad (R) \ (x \triangleleft y) \triangleleft = x \triangleleft \triangleleft y \)

Proof  \( (L) \) By NIND on \( y \): \( \Rightarrow(\varepsilon \triangleright x) = \varepsilon \triangleright \triangleright x, \Rightarrow(\xi y \triangleright x) = \Rightarrow(\xi y \triangleright x) = \Rightarrow(y \triangleright \triangleright x) = iy \triangleright \triangleright x. \) \( \square \)

Claim 3.2.7  \( (L) \ zy \triangleright zx = y \triangleright x \quad (R) \ xz \triangleleft yz = x \triangleleft y \)

Proof  \( (L) \) By NIND on \( z \): \( \varepsilon y \triangleright \varepsilon x = y \triangleright x \), \( \Rightarrow (ix) y \triangleright (ix) x = \Rightarrow (iy) \triangleright (ix) x = zy \triangleright zx = y \triangleright x. \) \( \square \)

Corollary 3.2.8  \( (L) \ x \triangleright xy = y \quad (R) \ y = yx \triangleleft x \)

Corollary 3.2.9  \( (L) \ x \triangleright x = \varepsilon \quad (R) \ \varepsilon = x \triangleleft x \)

Now, we are ready to define two more functions that will also be used as a convenient shorthand notation for the rest of the chapter.

Definition 3.2.2  \( (L) \ ′ = [\lambda x. x \triangleright \triangleright x] \quad (R) \ ′ = [\lambda x. x \triangleleft \triangleleft x] \)

(To make the notation more consistent with previous usage, we will write "\( a(x) \)" and "\( x^a \)" instead of the more formal "\( a(x) \)" and "\( x^a \)", respectively.)

Claim 3.2.10  \( (L) \ ′ \varepsilon = \varepsilon ∧ ′(0x) = 0 ∧ ′(1x) = 1 \quad (R) \ ′ \varepsilon = \varepsilon ∧ (x0)′ = 0 ∧ (x1)′ = 1 \)

Proof  \( (L) \) From the definition, by Claim 3.2.4 and by Corollary 3.2.8: \( ′ \varepsilon = \varepsilon ∧ ′x = \varepsilon ∧ x = \varepsilon, ′(ix) = ix ∧ ′(ix) = ix ∧ x = i. \) \( \square \)

Claim 3.2.11  \( (L) \ ′ x \cdot \triangleright x = x \quad (R) \ x = x ∧ x′ \)

Proof  \( (L) \) By NIND on \( x \): \( ′ x \cdot \varepsilon = \varepsilon \cdot \varepsilon = \varepsilon, ′(ix) \cdot \triangleright (ix) = i \cdot x = ix. \) \( \square \)

On "\( .^\wedge\)"

Claim 3.2.12  \( (L) \ \varepsilon \neq ix \quad (R) \ xi \neq \varepsilon \)

Proof  \( (L) \) By Axioms 2 and 3b and by taking the contrapositive: \( xi = \varepsilon \rightarrow x = \varepsilon ∧ i = \varepsilon \rightarrow i = \varepsilon. \) \( \square \)

Claim 3.2.13  \( (L) \ 0x \neq 1y \quad (R) \ 0x \neq y1 \)

Proof  \( (L) \) By Axiom 2 Claim 3.2.10 Axiom 1d and by taking the contrapositive: \( 0x = 1y \rightarrow ′(0x) = ′(1y) \rightarrow 0 = 1. \) \( \square \)
3.2. Developing the theory

Claim 3.2.14  \( (L) \ x = \varepsilon \implies x = \varepsilon \lor x = 0 \lor x = 1 \quad (R) \ x = \varepsilon \lor x = 0 \lor x = 1 \iff \varepsilon = x < \)

Proof  (L) By NIND on \( \varepsilon = \varepsilon \implies \varepsilon = \varepsilon \), \( (ix) = \varepsilon \implies x = \varepsilon \implies ix = i \varepsilon \iff ix = i \)  \( \Box \)

Theorem 3.2.1  \( (L) \ x = \varepsilon \lor x = 0 \lor x = 1 \quad (R) \ x = \varepsilon \lor x = x \iff 0 \lor x = x \iff 1 \)

Proof  (L) By NIND on \( \varepsilon = \varepsilon \), \( ix = i \cdot (ix) \).  \( \Box \)

Note that by Claims 3.2.12 and 3.2.13, we can show that exactly one of the disjuncts holds \( i.e. \), that \( x = \varepsilon \implies x \neq 0 \cdot \varepsilon \lor x \neq 1 \cdot \varepsilon \) and \( x = 0 \cdot \varepsilon \implies x \neq \varepsilon \land x \neq 1 \cdot \varepsilon \) and \( x = 1 \cdot \varepsilon \implies x \neq 0 \cdot \varepsilon \land x \neq \varepsilon \).

Corollary 3.2.15  \( (L) \ x \neq \varepsilon \implies x = 0 \cdot \varepsilon \lor x = 1 \cdot \varepsilon \quad (R) \ x \neq \varepsilon \implies x = x \iff 0 \lor x = x \iff 1 \)

Note that Theorem 3.2.1 can easily be generalized to show, for example, \( x = \varepsilon \lor x = 0 \lor x = 1 \lor x = 0 \cdot \varepsilon \lor x = 1 \cdot \varepsilon \lor x = 11 \cdot \varepsilon \lor x = \varepsilon \), or, by substituting various terms for \( x, ix = i \cdot x, ix = i \cdot (>x)i \lor xi = 1 \cdot (>x)i, etc. Because (A \lor B) \land (A \to C) \land (B \to C) \to C \) is a

Theorem of \( T_1 \), we can use Theorem 3.2.1 together with substitution to define an entire family of “derived rules” in \( T_1 \), like the following.

Derived Rule 3.2.1

1. \( (L) \ A[\varepsilon], A[0x], A[1x] \vdash A \quad (R) \ A[\varepsilon], A[x0], A[x1] \vdash A \)

2. \( A[\varepsilon], A[0], A[1], A[0x0], A[0x1], A[1x0], A[1x1] \vdash A \)

As an example of application of Derived Rule 3.2.1, we prove the following simple claim.

Claim 3.2.16  \( (>x) \leq (>x) \)

Proof  By Derived Rule 3.2.1  \( (\varepsilon) < \varepsilon = (>\varepsilon), (>i) < \varepsilon = (>\varepsilon = (>i), (>ixj) < = (xj) < = x = (>ix) = (>ixj) < ). \( \Box \)

On “\( 0 \)” and “\( 1 \)”

Claim 3.2.17  \( 0(x \cdot y) = 0x \cdot 0y \)

Proof  By NIND on \( y : 0(x \cdot \varepsilon) = 0x = 0x \cdot \varepsilon = 0x \cdot 0 \varepsilon, 0(x \cdot 0yi) = 0(x \cdot 0y) \cdot 0 = 0x \cdot 0y \cdot 0 = 0x \cdot 0yi). \( \Box \)

Note that an identical theorem can be proved with \( 1x \) in place of \( 0x \). In what follows, we will often need to prove theorems in which “\( 0 \)” or “\( 1 \)” appear, where the particular function used does not matter. We will indicate this by using “\( j \)” to stand for either of the above functions.
Claim 3.2.18 \[ j \cdot j = j \cdot j \]

Proof By NIND on \( x \): \( j \varepsilon \cdot j = \varepsilon \cdot j = j \cdot \varepsilon = j \cdot j \varepsilon, j(xi) \cdot j = (j \cdot j) \cdot j = j \cdot (j \cdot j) = j \cdot j(xi) \). \( \square \)

Claim 3.2.19 \( j \cdot jy = jy \cdot jx \)

Proof By NIND on \( y \), and by Claims 3.2.17 and 3.2.18: \( j \cdot j \varepsilon = j \cdot \varepsilon = j \cdot j \cdot j \varepsilon, j(yi) \cdot j = jy \cdot jx = j \cdot jy \cdot jx = jy \cdot jx = j(yi) \cdot jx \). \( \square \)

Corollary 3.2.20 \( j(xy) = j(ay) \)

Claim 3.2.21 \( jx = j(0x) \) and \( jx = j(1x) \)

Proof (We will prove only the first property, the second one being almost identical.) By NIND on \( x \): \( j \varepsilon = j(0 \varepsilon), j(xi) = jx \cdot j = j(0x) \cdot j = j(0x \cdot 0) = j(0(xi)) \). \( \square \)

Corollary 3.2.22 \( 0x = 0y \leftrightarrow 1x = 1y \)

Claim 3.2.23 \( \triangleright jx = jx \pretriangleright \)

Proof By NIND on \( x \), and by Claim 3.2.18: \( \triangleright j \varepsilon = \triangleright \varepsilon = \varepsilon \pretriangleright = j \varepsilon \pretriangleright, \triangleright j(xi) = \triangleright (j \cdot j) = \triangleright (j \cdot jx) = jx \cdot j = jx \pretriangleright = j(xi) \pretriangleright \). \( \square \)

Claim 3.2.24 (L) \( \triangleright (jx) = j(\triangleright x) \) (R) \( j(\pretriangleright x) = (jx) \pretriangleright \)

Proof (L) By NIND on \( x \), and by Claim 3.2.17: \( \triangleright j \varepsilon = \triangleright \varepsilon = \varepsilon = j \varepsilon = j(\triangleright \varepsilon), \triangleright j(ix) = \triangleright (j \cdot jx) = jx = j(\triangleright (ix)) \). \( \square \)

Claim 3.2.25 \( x = \varepsilon \leftrightarrow jx = \varepsilon \)

Proof By NIND on \( x \): \( \varepsilon = \varepsilon \leftrightarrow j \varepsilon = \varepsilon, xi = \varepsilon \leftrightarrow jx \cdot j = \varepsilon \). \( \square \)

Claim 3.2.26 (L) \( x \triangleright y = jx \triangleright y \) (R) \( y \pretriangleright x = y \pretriangleright jx \)

Proof (L) By NIND on \( x \): \( \varepsilon \triangleright y = y = j \varepsilon \triangleright y, ix \triangleright y = \triangleright (x \triangleright y) = \triangleright (j \cdot jx \triangleright y) = j \cdot jx \triangleright y = j(\triangleright (ix) \triangleright y) \). \( \square \)

Claim 3.2.27 (L) \( x \triangleright jy = j(x \triangleright y) \) (R) \( jy \pretriangleright x = j(y \pretriangleright x) \)

Proof (L) By NIND on \( x \), and by Claim 3.2.23: \( \varepsilon \triangleright jy = jy = j(\varepsilon \triangleright y), ix \triangleright jy = \triangleright (x \triangleright jy) = \triangleright j(x \triangleright y) = j(\triangleright (x \triangleright y)) = j(\triangleright (ix) \triangleright y) \). \( \square \)
3.2. Developing the theory

Claim 3.2.28 \( j(x > y) = j(y < x) \)

Proof By NIND on \( x \), and by Claims \[3.2.24\] and \[3.2.23\], \( j(\epsilon > y) = jy = j(y < \epsilon), j(ix > y) = j((x > y)) = j((x < y)) = j((y < x)) = j((y < x)) < = j((y < x)) = j(y < ix). \)

Corollary 3.2.29 \( x > jy = jy < x \)

Claim 3.2.30 \( (L) \) \( x = x < x' \) \( (R) \) \( x > x = x' \)

Proof \( (L) \) By NIND on \( x \), and by Corollary \[3.2.28\] and Claims \[3.2.26\] and \[3.2.23\], \( \epsilon = \epsilon = \epsilon < \epsilon, \epsilon(x) = i = ix < x = ix < 0x = ix < \epsilon > (0x) = ix < 0(x) = ix < 0(x) < = ix < (ix) <. \)

Corollary 3.2.31 \( (L) \) \( (xi) = xi < x \) \( (R) \) \( x > ix = (ix)' \)

Claim 3.2.32

1. \( (L) \) \( y > xi \neq \epsilon \leftrightarrow y > xi = y > x \cdot i \) \( (R) \) \( \epsilon \neq ix < y \leftrightarrow i \cdot x < y = ix < y \)

2. \( (L) \) \( y > x \neq \epsilon \rightarrow y > xi \neq \epsilon \) \( (R) \) \( \epsilon \neq x < y \rightarrow \epsilon \neq ix < y \)

3. \( (L) \) \( o(y \cdot (y > x)) = o(x \cdot (x > y)) \) \( (R) \) \( o((x < y) \cdot y) = o((y < x) \cdot x) \)

Proof

1. \( (L) \) The first direction is proved by Claim \[3.2.12\], \( y > xi = y > x \cdot i \rightarrow y > xi \neq \epsilon \). The other direction is proved by NIND on \( y \): \( \epsilon > xi \neq \epsilon \rightarrow \epsilon > xi = \epsilon > x \cdot i, jy > xi \neq \epsilon \rightarrow \epsilon > y > xi \neq \epsilon \rightarrow y > xi = y > x \cdot i \rightarrow \epsilon > (y > xi) = \epsilon > (y > x \cdot i) = y > x \cdot i \).

2. \( (L) \) By NIND on \( y \) and the preceding claim: \( \epsilon > x = x \neq \epsilon \rightarrow \epsilon > xi = xi \neq \epsilon, jy > x = \epsilon > (y > x) \neq \epsilon \rightarrow y > x \neq \epsilon \rightarrow y > xi \neq \epsilon \rightarrow y > xi = y > x \cdot i \rightarrow \epsilon > (y > xi) = \epsilon > (y > x) \cdot i \).

3. \( (L) \) The claim is proved first under the assumption that \( x > y = \epsilon \) (which implies by Axioms \[4c\] and \[5c\] and by the preceding claims, that \( y > xi = y > x \cdot i \), and also implies that \( xi > y = \epsilon \)) , and then under the assumption that \( x > y \neq \epsilon \) (which implies by Axioms \[4c\] and \[5c\] and by the preceding claims, that \( y > xi = y > x = \epsilon \)). Then, a simple application of modus ponens with the tautology \( x > y = \epsilon \land x > y \neq \epsilon \) yields the claim.

By NIND on \( x \), and under the assumption that \( x > y = \epsilon: o(y \cdot (y > x)) = o(y) = o(\epsilon \cdot (\epsilon > y)), o(y \cdot (y > xi)) = o(y \cdot (y > x) \cdot i) = o(y \cdot (y > x)) \cdot i = o(x \cdot (x > y)) \cdot 0 = o(x \cdot (x > y)) \cdot 0 = o(x \cdot 0) = o(xi \cdot \epsilon) = o(xi \cdot (x > y)) \)

By NIND on \( x \), and under the assumption that \( x \triangleright y \neq \varepsilon : 0(y \cdot (y \triangleright \varepsilon)) = 0y = 0(\varepsilon \cdot (\varepsilon \triangleright y)) \), 
\( 0(y \cdot (y \triangleright x)) = 0(y \cdot \varepsilon) = 0(y \cdot (y \triangleright x)) = 0(x \cdot (x \triangleright y)) = 0x \cdot 0(x \triangleright y) = 0x \cdot 0 \cdot 0(x \triangleright y) = 
0(xi) \cdot 0(xi \triangleright y) = 0(xi \cdot (xi \triangleright y)). \)

**On “?” and related functions**

Now, we will prove a group of theorems about the conditional function “?”\( ^{\text{?}} \). Note that in the statement of some of the theorems below, we will need to express the fact that terms \( t \) and \( u \) have the same length, something which can be done by the equation \( j_t = j_u \).

First, we introduce two new functions defined in terms of “?” that will be used throughout the rest of this chapter for notational convenience. Whereas the conditional function “?” performs a three-way test on its first argument, the “zero-length conditional” function “\( ?^{ZL} \)” tests whether the length of its first argument is zero or not, and the “even-length conditional” function “\( ?^{EL} \)” tests whether the length of its first argument is even or odd (\( ?^{EL} \) has already been introduced informally in Axiom 9).

**Definition 3.2.3** \( ?^{ZL} = [\lambda xyz. x ? (y, z, z)] \)

**Definition 3.2.4** \( ?^{EL} = [\lambda xyz. (x ▽ ▶ ▶) ?^{ZL} (y, z)] \)

(To make the notation more consistent with previous usage, we will write “\( x ?^{ZL} (y, z) \)” and “\( x ?^{EL} (y, z) \)” instead of the more formal “\( w ?^{ZL} (x, y, z) \)” and “\( w ?^{EL} (x, y, z) \)”, respectively.)

**Claim 3.2.33** \( w ? (x, y, z) = w ? (x, 0(z ▪ y) \cdot y, 0(y ▪ z) \cdot z) \)

**Proof** Immediate from Axiom 7 \( \square \)

**Claim 3.2.34** \( w ? (x, y, z) = w' ? (x, y, z) \)

**Proof** By NIND on \( w: \varepsilon ? (x, y, z) = x = \varepsilon' ? (x, y, z), w0 ? (x, y, z) = 0(z ▪ y) \cdot y = 0 ? (x, y, z) = (w0)' ? (x, y, z), w1 ? (x, y, z) = 0(y ▪ z) \cdot z = 1 ? (x, y, z) = (w1)' ? (x, y, z). \)

**Claim 3.2.35** \( wi ?^{ZL} (x, y) = y \)

**Proof** By Corollary 3.2.8: \( wi ?^{ZL} (x, y) = wi ? (x, y, y) = 0(y ▪ y) \cdot y = 0\varepsilon \cdot y = y. \)

**Corollary 3.2.36** \( iw ?^{ZL} (x, y) = y \)

**Corollary 3.2.37** \( w ?^{ZL} (x, y) = jw ?^{ZL} (x, y) \)

**Corollary 3.2.38** \( w ?^{ZL} (y, y) = w ? (y, y, y) = y \)
3.2. Developing the theory

**Theorem 3.2.2** For any k-ary function symbol f,

\((0y_1 = 0z_1 \land \cdots \land 0y_k = 0z_k) \rightarrow w(f(\bar{x}), f(\bar{y}), f(\bar{z})) = f(w(x_1, y_1, z_1), \ldots, w(x_n, y_n, z_n))\).

**Proof** By NIND on w, and under the assumption that \(0y_1 = 0z_1 \land \cdots \land 0y_k = 0z_k\),

\((f(\bar{x}), f(\bar{y}), f(\bar{z})) = f(\bar{x}) = f(\bar{y}) = f(\bar{z}) = f(w(x_1, y_1, z_1), \ldots, w(x_n, y_n, z_n))) = w(x_1, y_1, z_1), \ldots, w(x_n, y_n, z_n))\). □

**Claim 3.2.39**

\(w(x_1, y_1, z_1) = w(x_2, y_2, z_2) = w(x_3, y_3, z_3) = (x_1, 0(z_2 < y_2) \cdot y_2, 0(y_3 < z_3) \cdot z_3)\)

**Proof** By NIND on w and by Claim 3.2.39, \(\varepsilon(x_1, y_1, z_1) = \varepsilon(x_1, y_2, z_2) = \varepsilon(x_3, y_3, z_3)\) = \(\varepsilon(0(z_2 < y_2) \cdot y_2, 0(y_3 < z_3) \cdot z_3)\), \(w(x_1, y_1, z_1) = w(x_2, y_2, z_2) = w(x_3, y_3, z_3)\), and similarly for w1. □

**Claim 3.2.40**

\(w(x_0, y_0, z_0) = w(x_1, y_1, z_1) = w(x_2, y_2, z_2) = w(x_3, y_3, z_3)\)

**Proof** A straightforward NIND on w, very similar to the proof of Claim 3.2.39. □

**Corollary 3.2.41**

\(w^{2L}(x_0, y_0) = w^{2L}(x_1, y_1) = w^{2L}(x_2, y_2) = w^{2L}(x_3, y_3)\)

**Claim 3.2.42**

\(x^{2L}(y^{2L}(z_0, w_0), y^{2L}(z_1, w_1)) = x^{2L}(y^{2L}(z_0, w_0), x^{2L}(w_0, w_1))\)

**Proof** By NIND on x: \(\varepsilon^{2L}(y^{2L}(z_0, w_0), y^{2L}(z_1, w_1)) = y^{2L}(z_0, w_0) = y^{2L}(z_1, w_1) = y^{2L}(x^{2L}(z_0, z_1), x^{2L}(w_0, w_1))\). □

**Claim 3.2.43**

\(w = w \cdot (\varepsilon, w \cdot 0, w \cdot 1)\)

**Proof** By NIND on w: \(\varepsilon = \varepsilon \cdot (\varepsilon, \varepsilon \cdot 0, \varepsilon \cdot 1), w_0 = w_0 \cdot (\varepsilon, w_0, w_1) = w_0 \cdot (\varepsilon, (w_0) \cdot 0, (w_0) \cdot 1), w_1 = w_1 \cdot (\varepsilon, w_0, w_1) = w_1 \cdot (\varepsilon, (w_1) \cdot 0, (w_1) \cdot 1).\) □

**Corollary 3.2.44**

\(w = w^{2L}(\varepsilon, w)\)

**Theorem 3.2.3**

\(w(x, y_0, y_1) = z \leftrightarrow (w = x \land z = y) \lor (w = w \cdot 0 \land 0(y_1 < y_0) \cdot y_0 = z) \lor (w = w \cdot 1 \land 0(y_0 < y_1) \cdot y_1 = z)\)
**Proof** By NIND on $w$: $\varepsilon \neq (x, y_0, y_1) = z \leftrightarrow x = z \leftrightarrow \varepsilon = \varepsilon \land x = z$, $w \neq (x, y_0, y_1) = z \leftrightarrow_0 (y_1 < y_0) \cdot y_0 = z \leftrightarrow w\cdot (w_0 \cdot \cdot \cdot \land (y_1 < y_0) \cdot y_0 = z$, $w_1 ? (x, y_0, y_1) = z \leftrightarrow_0 (y_0 < y_1) \cdot y_1 = z \leftrightarrow w_1 = (w_1) \cdot \cdot \cdot \land (y_0 < y_1) \cdot y_1 = z$.

**Corollary 3.2.45** $w ?^{2L} (x, y) = z \leftrightarrow (w = \varepsilon \land x = z) \lor (w \neq \varepsilon \land y = x)$

**Theorem 3.2.4** For any term $u$, $w ?^{2L} (u[\varepsilon/w], u) = u$.

**Proof** By induction on the structure of $u$: if $u = w$, then $w ?^{2L} (u[\varepsilon/w], u) = w ?^{2L} (\varepsilon, w) = w$ by Corollary 3.2.44 if $u = x \neq w$, then $w ?^{2L} (u[\varepsilon/w], u) = w ?^{2L} (x, x) = x$ by Corollary 3.2.38 if $u = f(t_1, \ldots, t_n)$, then $w ?^{2L} (u[\varepsilon/w], u) = w ?^{2L} (f(t_1[\varepsilon/w], \ldots, t_n[\varepsilon/w]), f(t_1, \ldots, t_n)) = f(w ?^{2L} (t_1[\varepsilon/w], t_1), \ldots, w ?^{2L} (t_n[\varepsilon/w], t_n)) = f(t_1, \ldots, t_n) = u$ by Theorem 3.2.2 and the induction hypothesis.

**Claim 3.2.46** (L) $\forall (zx) = z ?^{2L} (\forall x, \forall z)$ (R) $\forall (xz) = z ?^{2L} (\forall x, \forall z)$

**Proof** (L) By NIND on $z$, and by Corollary 3.2.36 $\forall (\varepsilon x) = \forall x = \varepsilon ?^{2L} (\forall x, \forall x)$, $(\forall (\varepsilon) x) = \varepsilon ?^{2L} (\forall x, \forall (\varepsilon) x)$.

**Claim 3.2.47** (L) $\forall (zx) = z ?^{2L} (\forall x, \forall z \cdot x)$ (R) $\forall (xz) = z ?^{2L} (\forall x, \forall z \cdot x)$

**Proof** (L) By NIND on $z$, and by Corollary 3.2.36 $\forall (\varepsilon x) = \forall x = \varepsilon ?^{2L} (\forall x, \forall (\varepsilon) x)$, $(\forall (\varepsilon) x) = \varepsilon ?^{2L} (\forall x, \forall (\varepsilon) x)$. 

**Claim 3.2.48** $v ?^{2L} (u, t) = u \leftrightarrow (v \neq \varepsilon \rightarrow t = u)$

**Proof** By NIND on $v$: $\varepsilon ?^{2L} (u, t) = u \leftrightarrow u = u \leftrightarrow (\varepsilon \neq \varepsilon \rightarrow u = u)$, $vi ?^{2L} (u, t) = u \leftrightarrow t = u \leftrightarrow (vi \neq \varepsilon \rightarrow t = u)$.

**On “$<$” and “$>$”**

**Claim 3.2.49** (L) $j (x \bullet) = (j x) \bullet$ (R) $j (\bullet x) = \bullet (j x)$

**Proof** (L) By NIND on $x$, Axioms 9 and various theorems proved above: $j (\varepsilon \bullet) = \varepsilon = (j \varepsilon) \bullet$, $j ((i) \bullet) = j ((x \bullet \bullet) ?^{2L} (x \bullet, x \bullet \cdot \cdot \cdot (\bullet x \cdot i))) = j (x \bullet \bullet) ?^{2L} (j (x \bullet), j (x \bullet \cdot \cdot \cdot (\bullet x \cdot i))) = j (x \bullet) ?^{2L} (j (\bullet x), j (x \bullet \cdot \cdot \cdot (\bullet x \cdot i))) = j (x \bullet) ?^{2L} (j (x) \bullet, j (x) \bullet \cdot \cdot \cdot (\bullet (j x) \cdot j)) = (j x \cdot j) \bullet = (j (x i)) \bullet$. 

---
Basic properties of \("\text{EL}^*\) can easily be obtained from the basic properties of \("\text{ZL}\), on which it is based. In order to prove properties particular to \("\text{EL}^*\), we will need the following lemmas.

But first, a few reminders.

\[
\begin{align*}
\bullet \quad j(\triangleright x \cdot i)) &= \triangleright (\triangleright j x \cdot j) = \triangleright (j \cdot \triangleright j x) = j(\triangleright x) \\
\bullet \quad (z \cdot i) \triangleright y &= j(z \cdot i) \triangleright y = (j \cdot j z) \triangleright y = \triangleright (j z \triangleright y) = \triangleright (z \triangleright y)
\end{align*}
\]

**Lemma 3.2.50** \(0 x \triangleright (\triangleright 0 x \cdot 0) = (0 x \triangleright \triangleright 0 x) \cdot 0\)

**Proof** By Axioms 4c, 5c, and 8b, and by Claim 3.2.32, \(0 x \triangleright (\triangleright 0 x \cdot 0) = (0 x \triangleright \triangleright 0 x) \cdot 0\).

**Lemma 3.2.51** \(0 (xi) \triangleright \triangleright 0 (xi) = (0 x \triangleright \triangleright 0 x) \oplus (0 x \cdot 0)\)

**Proof** By Lemma 3.2.50,

\[
\begin{align*}
0 (xi) \triangleright \triangleright 0 (xi) &= x \text{EL}(0 x \triangleright (\triangleright 0 x \cdot 0)), (0 x \cdot 0) \triangleright (\triangleright 0 x \cdot 0) = \triangleright 0 x \cdot 0) \\
&= x \text{EL}(0 x \cdot 0) \triangleright 0 x, (0 x \cdot 0) \triangleright 0 x \\
&= (x \cdot 0) \text{EL}(0 x \cdot 0) \triangleright 0 x \\
&= (0 x \cdot 0) \text{EL}(0 x \cdot 0) \triangleright 0 x
\end{align*}
\]

**Theorem 3.2.5** \(x \text{EL}(y, z) = x \text{EL}(z, y)\)

**Proof** By Lemmas 3.2.50 and 3.2.51,

\[
\begin{align*}
x \text{EL}(y, z) &= 0 ( xi) \text{EL}(y, z) \\
&= (0 ( xi) \triangleright \triangleright 0 ( xi)) \text{EL}(y, z) \\
&= (0 x \cdot 0) \text{EL}(0 x \cdot 0) \triangleright 0 x, (0 x \cdot 0) \triangleright 0 x \\
&= (0 x \cdot 0) \text{EL}(0 x \cdot 0) \triangleright 0 x \\
&= (0 x \cdot 0) \text{EL}(0 x \cdot 0) \triangleright 0 x
\end{align*}
\]

**Claim 3.2.52** \((L) \quad (ixj) \cdot i = i \cdot x \triangleright \quad (R) \quad \triangleright (ixj) = \triangleright x \cdot j\)
Proof (L) By Axioms 3.2.5 and Theorem 3.2.5:

\[(ixj) ≜ xj \uparrow EL ((i(xj) ▶, i(xj) ▶)) = x \uparrow EL (i(xj) ▶, (i(xj) ▶) ▶) = x \uparrow EL (i \cdot x ▶, (i \cdot x ◄ ▶) ▶) = x \uparrow EL (i \cdot x ▶, i \cdot x ▶) = i \cdot x ▶. \]

Claim 3.2.53 (L) \((\triangleright x ◄) = \triangleright (x ▶)\) (R) \(\triangleright (\triangleright x ◄) = (\triangleright x) ▶\)

Proof (L) By Derived Rule 3.2.1 and Claim 3.2.52: \((\triangleright x ◄) = (\triangleright x) ▶ = \triangleright (i ▶) = \triangleright (i \cdot x ◄) = \triangleright (ixj) ▶. \)

3.2.2 Further definitions and theorems

In this section, we define many functions in \(T_1\) and prove their basic properties. We also give (and prove) a number of useful derived rules for \(T_1\).

From now on, we will not give proofs that consist only in a straightforward application of NIND. Proof sketches will be given for more complex theorems, and complete proofs are provided in Appendix A for most of the theorems below.

On generalizations of NIND

First, we define some generalizations of NIND based on Derived Rule 3.2.1.

Derived Rule 3.2.2

(L) \(A[ε], A[0], A[1], A[x] \rightarrow A[00x] \land A[01x] \land A[10x] \land A[11x] \vdash A\)

(R) \(A[ε], A[0], A[1], A[x] \rightarrow A[x00] \land A[x01] \land A[x10] \land A[x11] \vdash A\)

Proof (We will prove only (R), the case for (L) being almost identical.) Let us define a formula \(EL[x] : 0(x ▶ ▶ ▶ x) = ε\). By Lemma 3.2.51, we immediately get that \(EL[x] \leftrightarrow \neg EL[x] \leftrightarrow EL[ix].\) To prove that \(A\) is true under the given hypotheses, we will show that the hypotheses imply the following two statements:

\[(EL[x] \rightarrow A[x]) \land (\neg EL[x] \rightarrow A[x ◄]), \]
\[(\neg EL[x] \rightarrow A[x]) \land (EL[x] \rightarrow A[x ◄]).\] (3.2.1) (3.2.2)

Together with the fact that \(EL[x] \lor \neg EL[x]\), this will imply that \(A[x] \land A[x ◄], i.e., A[x] \) is true.
We can prove statement 3.2.1 by NIND on $x$: $(EL[ε] → A[ε]) ∧ (¬EL[ε] → A[ε])$ is trivially true since $A[ε]$ is true by assumption, while

$$(EL[xi] → A[xi]) ∧ (¬EL[xi] → A[(xi)] ↓) → (¬EL[x] → A[xi]) ∧ (EL[x] → A[x])$$

is true since the second conjunct is true by the induction hypothesis, and so is $(¬EL[x] → A[x])$, which, together with the assumption that $A[x] → A[xji]$, implies that $A[x] → A[xji] → A[xi]$. The same reasoning applies to statement 3.2.2, which concludes the proof. □

Note that this proof can easily be modified to get a similar derived rule for $A[x] → A[ixi]$, and it can easily be extended to cover other variations of Theorem 3.2.1.

Next, we want to define simultaneous NIND on two variables. Before we can do this, we need to define a few functions and prove their basic properties.

**Definition 3.2.5** (L) $lb = [\lambda xy. y ?^L (ε, (\triangleright y \triangleright x))]$ (R) $rb = [\lambda xy. y ?^L (ε, (x < y < y'))$

**Definition 3.2.6** (L) $lc = [\lambda xy. x < (y ↓ x)]$ (R) $rc = [\lambda xy. (x < y) ↓ x]$

**Definition 3.2.7** $\min^t = [\lambda xy. x < y ?^L (x, y)]$ $\max^t = [\lambda xy. x ↑ y ?^L (x, y)]$

**Claim 3.2.54**

1. (L) $z \triangleright yx = z \triangleright y \cdot (z < y) \triangleright x$ (R) $xy \triangleright z = x \triangleright (y \triangleright z) \cdot y < z$

2. (L) $y < ↓ x = lb(x, y) \cdot y \triangleright x$ (R) $x < ↑ y = x \triangleright y \cdot rb(x, y)$

3. (L) $y \triangleright ((x < y) \triangleright x) = ε$ (R) $(x < (y \triangleright x)) \triangleright y = ε$

4. (L) $lc(rc(x, y), y) = rc(x, y)$ (R) $rc(lc(x, y), y) = lc(x, y)$

5. (L) $(y \triangleright x) = y \triangleright x ?^L (ε, (x < (y \triangleright x)))$ (R) $(x < y)' = x < y ?^L (ε, ((x < y) \triangleright x))$

6. (L) $lc(x, yi) = lc(x, y) \cdot lb(x, yi)$ (R) $rc(x, iy) = rb(x, iy) \cdot rc(x, y)$

7. (L) $lc(x, y) \cdot y \triangleright x = x$ (R) $x = x < y \cdot rc(x, y)$

Now, we can state and prove a derived rule for simultaneous NIND on two variables.

**Derived Rule 3.2.3**

(LL) $A[ε, y], A[x, ε], A[x, y] → A[0x, 0y] \land A[0x, 1y] \land A[1x, 0y] \land A[1x, 1y] \vdash A$

(LR) $A[ε, y], A[x, ε], A[x, y] → A[0x, y0] \land A[0x, y1] \land A[1x, y0] \land A[1x, y1] \vdash A$

(RL) $A[ε, y], A[x, ε], A[x, y] → A[x0, 0y] \land A[x0, 1y] \land A[x1, 0y] \land A[x1, 1y] \vdash A$

(RR) $A[ε, y], A[x, ε], A[x, y] → A[x0, y0] \land A[x0, y1] \land A[x1, y0] \land A[x1, y1] \vdash A$
Chapter 3. The Quantifier-Free Theory $T_1$

Proof: (We will prove only (RR), the other cases being almost identical.) Under the given assumptions, we will prove $A[x_L \cdot lc(x_R, z), y_L \cdot lc(y_R, z)]$ by NIND on $z$, where

$$
x_L = x < \min^L(x, y) \quad x_R = rc(x, \min^L(x, y))
$$
$$
y_L = y < \min^L(x, y) \quad y_R = rc(y, \min^L(x, y))
$$

Base case: $A[x_L \cdot lc(x_R, \varepsilon), y_L \cdot lc(y_R, \varepsilon)] = A[x_L, y_L]$. By the definition of $\min^L$, we know that $\min^L(x, y) = x \vee \min^L(x, y) = y$, which means that $x_L = \varepsilon \vee y_L = \varepsilon$, which implies $(A[x_L, y_L] \leftrightarrow A[\varepsilon, y_L])$, so we know that $A[x_L, y_L]$ is true by the assumptions.

Induction Step: we have that

$$
A[x_L \cdot lc(x_R, zi), y_L \cdot lc(y_R, zi)] = A[x_L \cdot lc(x_R, z) \cdot lb(x_R, zi), y_L \cdot lc(y_R, z) \cdot lb(y_R, zi)],
$$

which follows directly from the induction hypothesis by the assumptions.

Finally, we know that $A[x_L \cdot lc(x_R, \min^L(x, y)), y_L \cdot lc(y_R, \min^L(x, y))] = A[x_L \cdot x_R, y_L \cdot y_R] = A[x, y]$, which completes the proof. \(\square\)

Note that this rule, and its proof, can easily be extended to more than two variables, giving us a very useful form of NIND on many variables.

On propositional reasoning

Now, we will show how to formalize propositional connectives in $T_1$. (The definitions are identical to those for $L_1$, and we use "$x \approx_B (y, x)"$ instead of the more formal "$\approx_B(x, y, z)$".)

Definition 3.2.8

1. $\approx_B = [\lambda xyz, x ? (z, z, y)]$
2. $\approx_B = [\lambda x. x \approx_B (1, 0)]$
3. $\neg_B = [\lambda x. x \approx_B (0, 1)]$
4. $\wedge_B = [\lambda xy. x \approx_B (\approx_B y, 0)]$
5. $\vee_B = [\lambda xy. x \approx_B (1, \approx_B y)]$
6. $\rightarrow_B = [\lambda xy. x \approx_B (\approx_B y, 1)]$
7. $\leftrightarrow_B = [\lambda xy. x \approx_B (\approx_B y, \neg_B y)]$
8. $\oplus_B = [\lambda xy. x \approx_B (\neg_B y, \approx_B y)]$

The properties of "$\approx_B"$ already proven immediately extend to $\approx_B$ in the obvious way, and the following theorem follows directly from these properties.
Theorem 3.2.6

1. \( \approx^B x = 1 \lor \approx^B x = 0 \)

2. \( \approx^B \approx^B x = \approx^B x \)

3. \( \neg^B x = 1 \leftrightarrow \neg(\approx^B x = 1) \)

4. \( x \land^B y = 1 \leftrightarrow (\approx^B x = 1 \land \approx^B y = 1) \)

5. \( x \lor^B y = 1 \leftrightarrow (\approx^B x = 1 \lor \approx^B y = 1) \)

6. \( x \rightarrow^B y = 1 \leftrightarrow (\approx^B x = 1 \rightarrow \approx^B y = 1) \)

7. \( x \leftrightarrow^B y = 1 \leftrightarrow (\approx^B x = 1 \leftrightarrow \approx^B y = 1) \)

8. \( x \oplus^B y = 1 \leftrightarrow (\approx^B x = 1 \oplus \approx^B y = 1) \)

This theorem gives us direct proofs of the usual properties of the defined connectives, from the corresponding properties of the connectives in \( T_1 \), and it allows us to introduce the following notation: we will write “\( t \)” instead of “\( t = 1 \)” for \( T_1 \)-terms \( t \). (For example, we could state that “\( \approx^B x \oplus^B \neg^B x \)” is a theorem.)

On variations of TRN

To define functions by “simple” TRN, we will use \( \text{STRN}[g, h] \) as shorthand for

\[
[\lambda x y . \text{TRN}[\lambda x z y . g(x, y), \lambda x z y v_\ell v_r . h(x, y, v_\ell, v_r) , \lambda z . z, \lambda z . z](x, \varepsilon, y)].
\]

The following property is then a direct consequence of the axiom for TRN.

Claim 3.2.55

\[
\text{STRN}[g, h](x, y) = x \approx^2 (g(x, y), h(x, y, \text{STRN}[g, h](x \bowtie, y)), \text{STRN}[g, h](\uplus x, y)).
\]

On “AND” and “OR”

Definition 3.2.9 \( \text{AND} = \text{STRN}[\lambda x . x, \lambda x v_\ell v_r . v_\ell \land^B v_r] \)

Definition 3.2.10 \( \text{OR} = \text{STRN}[\lambda x . x, \lambda x v_\ell v_r . v_\ell \lor^B v_r] \)

We can use TIND to prove the following simple theorem (we give the proof here to illustrate the use of TIND).

Theorem 3.2.7 \( \text{AND}(1x) = x \approx^2 (\varepsilon, 1) \)
Proof. By TIND on \( x \): \( \text{AND}(\epsilon) = \text{AND}(\epsilon) = \epsilon = \epsilon \land_{\text{TL}} (\epsilon, 1) \), \( \text{AND}(1) = \text{AND}(1) = 1 = i \land_{\text{TL}} (\epsilon, 1) \), \( \text{AND}(x) = \text{AND}((1)x) \land^B \text{AND}(\uparrow x)) = \text{AND}(1(x)) \land^B \text{AND}(\uparrow x) = x \land_{\text{TL}} (\epsilon, 1) \land^B \uparrow x \land_{\text{TL}} (\epsilon, 1) = 1 \land^B 1 = 1. \quad \Box 

Similarly, we can prove that \( \text{OR}(\epsilon x) = x \land_{\text{TL}} (\epsilon, 0) \).

Now, we want to prove some more basic properties of \( \text{AND} \) and \( \text{OR} \), the most important of which being \( \text{AND}(xy) = \text{AND}(x) \land^B \text{AND}(y) \) (and similarly for \( \text{OR} \)). This property would naturally be proved using NIND since it involves the concatenation of two variables, but \( \text{AND} \) is defined by TRN which makes it more natural to use TIND. In fact, we will use TIND to prove the property but because of the messy interaction between concatenation recursion and tree recursion, the proof will unfortunately not be as simple as one might expect.

Before we get started, note that it is a simple matter to extend Derived Rule 3.2.4 to give us rules similar to the following ones.

**Derived Rule 3.2.4**

\[ A[0], A[1], y \neq \epsilon \land A[y] \rightarrow A[y0] \land A[y1] \vdash x \neq \epsilon \rightarrow A[x] \]

**Derived Rule 3.2.5**

\[ A[00], A[01], A[10], A[11], A[000], \ldots, A[111], y \blacktriangleleft \epsilon \neq \epsilon \land A[y] \land A[\blacktriangleleft y] \rightarrow A[y] \]

\[ \vdash x \neq \epsilon \land x \neq 0 \land x \neq 1 \rightarrow A[x] \]

These rules can then be used to prove the following claim and theorem.

**Lemma 3.2.56**

1. (L) \( x \neq \epsilon \land x \neq 0 \land x \neq 1 \rightarrow (\blacktriangleleft x \cdot j) \blacktriangleleft x = (x \blacktriangleleft ) \cdot \blacktriangleleft x \)

   (R) \( x \neq \epsilon \land x \neq 0 \land x \neq 1 \rightarrow (\blacktriangleleft x \cdot j) = (\blacktriangleleft x \cdot j) \)

2. (L) \( \blacktriangleleft x \land^B \text{AND}(\blacktriangleleft x \cdot j) = \text{AND}(x) \land^B j \) \quad (R) \( j \land^B \text{AND}(x) = \text{AND}(j \cdot x) \land^B x' \)

3. \( \blacktriangleleft x \land^B \text{AND}(\blacktriangleleft x) = \text{AND}(x) = \text{AND}(x) \land^B x' \) for \( x \neq \epsilon, 0, 1 \)

**Theorem 3.2.8**

\( \text{AND}(xy) = \text{AND}(x) \land^B \text{AND}(y) \) for \( x, y \neq \epsilon \)

Proof. By Lemma 3.2.56 and by Derived Rule 3.2.4 on \( y \): \( \text{AND}(xi) = \text{AND}(x) \land^B i = \text{AND}(x) \land^B \text{AND}(i), \text{AND}(x(yi)) = \text{AND}(xy) \land^B i = \text{AND}(x) \land^B \text{AND}(y) \land^B i = \text{AND}(x) \land^B \text{AND}(yi) \). \( \Box \)

Note that a similar lemma and theorem can be used to show \( \text{OR}(xy) = \text{OR}(x) \lor^B \text{OR}(y) \) for \( x, y \neq \epsilon \).
3.2. Developing the theory

On generalizations of CRN—part I

Now, we will define simultaneous concatenation recursion on notation for many variables, prove its basic properties, and define a few more useful functions based on this generalized CRN. But first, we must prove a number of technical lemmas.

**Theorem 3.2.9**  
\( jx = jy \leftrightarrow x \triangleright y = \varepsilon = x \triangleleft y \)

**Claim 3.2.57**  
For \( f = rCRN[h] \),

1. \( j(f(x, \vec{y})) = jx \)
2. \( f(x, \vec{y}) \triangleleft z = f(x \triangleleft z, \vec{y}) \)
3. \( \lambda y. j(y \triangleright (x \triangleleft y)) = (0 \cdot h(lc(x, z), \vec{y}))', \varepsilon \) for \( x, z \neq \varepsilon \)
4. \( lc(f(x, \vec{y}), z) = f(lc(x, z), \vec{y}) \)

(A similar claim can be proved about \( \ell CRN \).)

**Definition 3.2.11**  
\( \text{(L)} \quad \lambda y. j(y \triangleright x) \cdot x \)  
\( \text{(R)} \quad \lambda y. x \cdot j(x \triangleright y) \)

**Definition 3.2.12**

\[
\begin{align*}
\text{max}_1^L &= [\lambda x.x] \\
\text{max}_{k+1}^L &= [\lambda x.\text{max}_k^L(x, \text{max}_k^L(\vec{x}_k))] \\
\end{align*}
\]

**Lemma 3.2.58**

1. \( \text{(L)} \quad y \triangleright x = x \triangleright y \quad \text{(R)} \quad x \triangleleft y = x \triangleright y \quad \text{for} \quad x, y \neq \varepsilon \)
2. \( \text{(L)} \quad (y \triangleleft (x \triangleright y)) \triangleright x = y \triangleright x \quad \text{(R)} \quad x \triangleleft ((y \triangleleft x) \triangleright y) = x \triangleleft y \)

**Claim 3.2.59**

1. \( \text{(L)} \quad \lambda y. j(x, \text{max}_k^L(x, y)) = \lambda y. j(x, y) \quad \text{(R)} \quad \lambda y. j(x, \text{max}_k^L(x, y)) = \lambda y. j(x, y) \)
2. \( \text{(L)} \quad \lambda y. j(x, \text{max}_k^L(x, y)) = i \quad \text{(R)} \quad \lambda y. j(x, \text{max}_k^L(x, y)) = i \)

(Note that this can easily be generalized to more than two variables.)

Now, we are ready to define \( \ell CRN_m \) and \( rCRN_m \).
Definition 3.2.13

\[ \ell \text{CRN}_m[h] = [\lambda x \bar{y}. \ell \text{CRN}[\lambda z x \bar{y}. h(rc(x_1, z), \ldots, rc(x_m, z), y)]] \]

\[ r \text{CRN}_m[h] = [\lambda x \bar{y}. r \text{CRN}[\lambda z x \bar{y}. h(lc(x_1, z), \ldots, lc(x_m, z), y)]] \]

A simple application of Derived Rule 3.2.3 together with Claim 3.2.59 both generalized to the \( m \) variables \( x_1, \ldots, x_m \), suffices to show the following basic theorem about \( \ell \text{CRN}_m \) and \( r \text{CRN}_m \).

Theorem 3.2.10

\[ \ell \text{CRN}_m[h](x_1, \ldots, x_m, y) = x_1 \cdots x_m \upmodels (\varepsilon, r \text{CRN}_m[h](r \text{p}_0(x_1, \max^L_m(\bar{x}_m)), \ldots, r \text{p}_0(x_m, \max^L_m(\bar{x}_m)), y))' \]

Using \( \ell \text{CRN}_m \), we can now define some useful functions and prove their basic properties.

Definition 3.2.14

1. \( \text{not}^B = \ell \text{CRN}[\lambda x. \neg^B(x)] \)
2. \( \text{and}^B_m = \ell \text{CRN}_m[\lambda x \bar{m}. (x_1) \land^B \cdots \land^B (x_m)] \)
3. \( \text{or}^B_m = \ell \text{CRN}_m[\lambda x \bar{m}. (x_1) \lor^B \cdots \lor^B (x_m)] \)
4. \( \text{xor}^B_m = \ell \text{CRN}_m[\lambda x \bar{m}. (x_1) \oplus^B \cdots \oplus^B (x_m)] \)
5. \( \text{iff}^B_m = \ell \text{CRN}_m[\lambda x \bar{m}. (x_1) \leftrightarrow^B (x_2) \land^B \cdots \land^B (x_{m-1}) \leftrightarrow^B (x_m)] \)

Claim 3.2.60

1. \( \text{and}^B_m(x_1 \bar{i}_1, \ldots, x_m \bar{i}_m) = \text{and}^B_m(\bar{x}_m) \cdot \text{and}^B_m(\bar{t}_m) \)
2. \( jx_1 = \cdots = jx_m \land jy_1 = \cdots = jy_m \rightarrow \text{and}^B_m(x_1 y_1, \ldots, x_m y_m) = \text{and}^B_m(\bar{x}_m) \cdot \text{and}^B_m(\bar{y}_m) \).
3. \( \text{and}^B_m(\bar{x}_m) \downarrow = \text{and}^B_m(x_1 \downarrow, \ldots, x_m \downarrow) \)
4. \( \text{and}^B_m(\bar{x}_m) \downarrow y = \text{and}^B_m(x_1 \downarrow y, \ldots, x_m \downarrow y) \)
3.2. Developing the theory

And similarly for \( or_m^B \), iff \( B_m \), and \( not^B \). We can now prove a theorem relating the functions \( AND \) and \( OR \) to each other.

**Theorem 3.2.11**

\[ \neg^B AND(x) = OR(not^B(x)) \quad \text{and} \quad \neg^B OR(x) = AND(not^B(x)) \quad \text{for } x \neq \varepsilon \]

**On generalizations of CRN—part II**

Finally, we are ready to generalize CRN to operate on blocks of bits instead of single bits. For technical reasons to be discussed below, this will be done only for variable-length blocks of bits whose lengths are powers of two.

In order to get such a version of CRN, we first need to define a length-division function. Ideally, we would like to define a function \( \text{div}^L(x, y) \) whose length would be equal to \( \lfloor |x|/|y| \rfloor \), but this seems to be impossible in \( T_1 \). Instead, we can define a function \( \text{powdiv}^L(x, y) \) whose length is equal to \( \lfloor |x|/2^{\lceil \lg |y| \rceil} \rfloor \), i.e., the function divides the length of \( x \) by the smallest power of 2 larger than or equal to the length of \( y \). Luckily, this will be sufficient for our purposes, as will be seen in Chapter 5.

The first functions we define are a function that returns a string whose length is the smallest power of two larger than or equal to the length of its input, and a function that tests whether or not the length of its input is a power of two.

**Definition 3.2.15**

\[ \text{pow}^L = \text{STRN}[1, \lambda xyv_r . v_r \cdot v_r] \]

**Definition 3.2.16**

\[ \text{ispow}^L = \text{STRN}[\lambda y . \varepsilon, \lambda xyv_r . v_r ?^{EL} (v_r, 1)] \]

(Note that \( \text{ispow}^L \) returns \( \varepsilon \) if the length of its input is a power of two and 1 otherwise.) The basic properties of \( \text{pow}^L \) and \( \text{ispow}^L \) are now easy to prove by TIND.

**Claim 3.2.61**

1. \( \triangleright \text{pow}^L(y) = \text{pow}^L(\triangleright y) = \text{pow}^L(y) \quad \text{(for } y \neq \varepsilon, 0, 1 \text{)} \)
2. \( 1 \text{pow}^L(y) = \text{pow}^L(1y) = \text{pow}^L(y) \)
3. \( \text{pow}^L(\text{pow}^L(y)) = \text{pow}^L(y) \)
4. \( \text{pow}^L(y) \triangleright y = \varepsilon \)
5. \( \text{ispow}^L(\text{pow}^L(y)) = \varepsilon \)

Before we can define the length-division function, we need to define a “length multiplication” function (this is just the “smash” function).

**Definition 3.2.17**

\[ \# = \text{STRN}[\lambda xy . x ?^{2L} (\varepsilon, y), \lambda xyv_r . v_r \cdot v_r] \]
These properties of \# can then be proven with simple applications of TIND and NIND.

\textbf{Claim 3.2.62}

1. $\varepsilon \# y = \varepsilon = x \# \varepsilon$

2. (L) $1(x \# y) = 1(x \# y) \cdot 1y$ \quad (R) $1(x \# yi) = 1(x \# y) \cdot 1x$

3. (L) $1((x \cdot y) \# z) = 1(x \# z) \cdot 1(y \# z)$ \quad (R) $1(x \# (y \cdot z)) = 1(x \# y) \cdot 1(x \# z)$

4. $1(x \# y) = 1(y \# x)$

Now, we can define the $\text{powdiv}^L$ function, and a corresponding $\text{powmod}^L$ function.

\textbf{Definition 3.2.18}

\[
\text{powdiv}^L = \text{TRN}[\lambda yx. y ?^{x}L (\varepsilon, 1x) \cdot \lambda yxv_r. v_r, \blacklozenge, \blacklozenge]
\]

\[
\text{powmod}^L = [\lambda xy. (\text{pow}^L(y) \# \text{powdiv}^L(x, y)) \triangleright 1x]
\]

Straightforward applications of TIND then prove these basic properties.

\textbf{Claim 3.2.63}

1. $\text{powdiv}^L(x, y) = \text{powdiv}^L(1x, 1y)$

2. $\text{powdiv}^L(x, y) = \text{powdiv}^L(x, \text{pow}^L(y))$

3. $x \triangleright \text{pow}^L(y) \neq \varepsilon \rightarrow \text{powdiv}^L(x, y) = \varepsilon$

4. $\text{powdiv}^L(\text{pow}^L(y) \# z, y) = y ?^{x}L (\varepsilon, 1z)$

And properties of $\text{powmod}^L$ follow directly from the properties of $\text{powdiv}^L$.

\textbf{Corollary 3.2.64}

1. $\text{powmod}^L(x, y) = \text{powmod}^L(1x, 1y)$

2. $\text{powmod}^L(x, y) = \text{powmod}^L(x, \text{pow}^L(y))$

3. $x \triangleright \text{pow}^L(y) \neq \varepsilon \rightarrow \text{powmod}^L(x, y) = 1x$

4. $\text{powmod}^L(\text{pow}^L(y) \# z, y) = \varepsilon$

We need just a few more technical lemmas about $\text{powdiv}^L$ and $\text{powmod}^L$ before we can define generalized CRN and prove its properties.
3.2. Developing the theory

Claim 3.2.65

1. \( x \triangleright (\text{pow}^k(y) \not\# \text{powdiv}^k(x, y)) = \varepsilon \)

2. \( y \neq \varepsilon \implies \text{powmod}^k(x, y) \triangleright \text{pow}^k(y) \neq \varepsilon \)

3. \( y \neq \varepsilon \implies \text{powdiv}^k(x1, y) = \text{powdiv}^k(x, y) \cdot ( (\text{powmod}^k(x, y) \cdot 1) \triangleright \text{pow}^k(y) \overset{2L}{=} (1, \varepsilon) ) \)

4. \( y \neq \varepsilon \implies \text{powmod}^k(x1, y) = (\text{powmod}^k(x, y) \cdot 1) \triangleright \text{pow}^k(y) \overset{2L}{=} (\varepsilon, \text{powmod}^k(x, y) \cdot 1) \)

5. \( y \neq \varepsilon \land x \triangleright \text{pow}^k(y) = \varepsilon \implies \text{powdiv}^k(x, y) = \text{powdiv}^k(x \triangleleft \text{pow}^k(y), y) \cdot 1 \)

At last, we are ready to define the generalized versions of \( \ell \text{CRN} \) and \( r \text{CRN} \) which we will name “\( \ell \text{powCRN} \)” and “\( r \text{powCRN} \)”. Functions defined by \( \ell \text{powCRN} \) or \( r \text{powCRN} \) take two extra parameters \( u, v \) as input, and essentially perform CRN on their first input by replacing blocks of \( |\text{pow}^k(u)| \) bits with blocks of \( |\text{pow}^k(v)| \) bits. Just like \((2^k, 2^k)\)-CRN in Chapter 2 we will simulate these generalized forms of CRN by using \( \text{powdiv}^k \) and \( \text{powmod}^k \) to extract the correct substring of the first input to pass to \( h \) and to output the correct bits of \( h \) in sequence. Intuitively, \( \ell \text{powCRN}[g, h] \) and \( r \text{powCRN}[g, h] \) will behave as follows (for all strings \( z \) such that \( |z| = |\text{pow}^k(u)| \)).

\[
\ell \text{powCRN}[g, h](x, u, v, \overline{y}) = g(x, u, v, \overline{y}) \quad (\text{if } |x| < |\text{pow}^k(u)|)
\]

\[
\ell \text{powCRN}[g, h](z \cdot x, u, v, \overline{y}) = \lambda a_0 \left( h(z \cdot x, u, v, \overline{y}), \text{pow}^k(v) \right) \cdot \ell \text{powCRN}[g, h](x, u, v, \overline{y})
\]

\[
r \text{powCRN}[g, h](x, u, v, \overline{y}) = g(x, u, v, \overline{y}) \quad (\text{if } |x| < |\text{pow}^k(u)|)
\]

\[
r \text{powCRN}[g, h](x \cdot z, u, v, \overline{y}) = r \text{powCRN}[g, h](x, u, v, \overline{y}) \cdot r a_0 \left( h(x \cdot z, u, v, \overline{y}), \text{pow}^k(v) \right)
\]

Where we have used the functions \( \lambda a_j \) and \( r a_j \), whose definitions and basic properties (easily proved by Derived Rule 3.2.31) appear below.

Definition 3.2.19

(L) \( \lambda a_j = [\lambda x y \cdot (y \triangleleft x) \cdot ((x \triangleleft y) \triangleright x)] \)  \quad (R) \( r a_j = [\lambda x y \cdot (x \triangleleft (y \triangleright x)) \cdot j(x \triangleright y)] \)

Claim 3.2.66

1. \( (L) \ 1(\lambda a_j(x, y)) = 1y \quad (R) \ 1y = 1(r a_j(x, y)) \)

2. \( (L) \ \lambda c(\lambda a_j(x, y), y) = \lambda a_j(x, y) \quad (R) \ r a_j(x, y) = r c(r a_j(x, y), y) \)
Definition 3.2.20

$$\ell_{\text{powCRN}}[g, h] = [\lambda xuvy. \ell_{\text{CRN}}[\lambda zuvy.]$$

$$\begin{align*}
rb & (la_0(h rc(x, (\text{pow}^l(u) \# (1 \cdot \text{powdiv}^l(\rangle z, v))) \cdot \text{powmod}^l(x, u)), u, v, \bar{y}), \\
& \text{pow}^l(v), 1 \cdot \text{powmod}^l(\rangle z, v)) (\text{pow}^l(v) \# \text{powdiv}^l(x, u), x, u, v, \bar{y}) \\
& \cdot g(rc(x, \text{powmod}^l(x, u)), u, v, \bar{y})
\end{align*}$$

$$rp_{\text{powCRN}}[g, h] = [\lambda xuvy. g(lc(x, \text{powmod}^l(x, u)), u, v, \bar{y}) \cdot r_{\text{CRN}}[\lambda zuvy.]$$

$$\begin{align*}
rb & (ra_0(h(lc(x, \text{powmod}^l(x, u)) \cdot (\text{pow}^l(u) \# (\text{powdiv}^l(z, v) \cdot 1)) \cdot u, v, \bar{y}), \\
& \text{pow}^l(v), \text{powmod}^l(z, v) \cdot 1)) (\text{pow}^l(v) \# \text{powdiv}^l(x, u), x, u, v, \bar{y})
\end{align*}$$

Theorem 3.2.12 For $$u \neq \varepsilon$$ and $$v \neq \varepsilon$$,

$$\ell_{\text{powCRN}}[g, h][x, u, v, \bar{y}] =$$

$$x \triangleright \text{pow}^l(u) \text{?zl} \left(la_0(h(x, u, v, \bar{y}), \text{pow}^l(v)) \cdot \ell_{\text{powCRN}}[g, h][\text{pow}^l(u) \triangleright x, u, v, \bar{y}], g(x, u, v, \bar{y})\right),$$

$$r_{\text{powCRN}}[g, h][x, u, v, \bar{y}] =$$

$$x \triangleright \text{pow}^l(u) \text{?zl} \left(r_{\text{powCRN}}[g, h][x \triangleleft \text{pow}^l(u), u, v, \bar{y}] \cdot ra_0(h(x, u, v, \bar{y}), \text{pow}^l(v)), g(x, u, v, \bar{y})\right).$$

Proof We prove the theorem for $$r_{\text{powCRN}}$$ only, the case for $$\ell_{\text{powCRN}}$$ being almost identical. To start with, if $$x \triangleright \text{pow}^l(u) \neq \varepsilon$$, then Claim 3.2.63 and Corollary 3.2.64 give us the result immediately since $$\text{powdiv}^l(x, u) = \varepsilon$$ and $$\text{powmod}^l(x, u) = 1_x$$.

Next, suppose that $$x \triangleright \text{pow}^l(u) = \varepsilon$$. Then, Claim 3.2.65 implies that $$\text{powdiv}^l(x, u) \neq \varepsilon$$, which shows that $$\text{pow}^l(v) \neq \text{powdiv}^l(x, u) = (\text{pow}^l(v) \# \text{powdiv}^l(x, u) \cdot 1) \cdot \text{pow}^l(v) = (\text{pow}^l(v) \# \text{powdiv}^l(x \triangleleft \text{pow}^l(u), u)) \cdot \text{pow}^l(v)$$. The following facts are then direct consequences of preceding claims, and hold for all strings $$z$$ such that $$z \neq \varepsilon \land z \triangleleft \text{pow}^l(v) = \varepsilon$$:

- $\text{lc}(x, y \cdot z) = \text{lc}(x, y) \cdot \text{lc}(y \triangleright x, z)$ (easy to prove by NIND on $$z$$),
- $\text{powmod}^l((\text{pow}^l(v) \# \text{powdiv}^l(x, u) \cdot 1z \triangleleft, \text{pow}^l(v)) \cdot 1 = \text{powmod}^l(1z \triangleleft, \text{pow}^l(v)) \cdot 1 = 1z$,
- $\text{lc} \left(x, \text{powmod}^l(x, u) \cdot (\text{pow}^l(u) \# (\text{powdiv}^l((\text{pow}^l(v) \# \text{powdiv}^l(x, u) \cdot 1z \triangleleft, v) \cdot 1)) \right)) = \text{lc} \left(x, \text{powmod}^l(x, u) \cdot (\text{pow}^l(u) \# (\text{powdiv}^l(x, u) \cdot 1)) \right) = \text{lc}(x, 1_x) = x$.

These facts can be used to prove by NIND on $$z$$ that

$$x \triangleright \text{pow}^l(u) = \varepsilon \land z \triangleleft \text{pow}^l(v) = \varepsilon \rightarrow f( (\text{pow}^l(v) \# \text{powdiv}^l(x, u) \cdot 1z, x, u, v, \bar{y}) =$$

$$f(\text{pow}^l(v) \# \text{powdiv}^l(x \triangleleft \text{pow}^l(u), u), x, u, v, \bar{y}) \cdot rc(ra_0(h(x, u, v, \bar{y}), \text{pow}^l(v)), z)$$

(where we use “$$f$$” to denote the function defined by $$r_{\text{CRN}}$$ in the definition of $$r_{\text{powCRN}}$$), and putting $$z = \text{pow}^l(v)$$ in this last fact gives us the theorem. \(\blacksquare\)
3.2. Developing the theory

3.2.3 Numerical definitions and theorems

In this section, we will give definitions for numerical predicates and functions (i.e., ones that treat their string arguments as encoding binary numbers) and prove their properties.

On “$=^N$” and “$<^N$”

The definitions of “$=^N$” and “$<^N$” inside $T_1$ are the same as in $L_1$. Intuitively, $x =^N y$ if the two strings are equal when padded on the left with 0’s to the same length.

**Definition 3.2.21**

$$=^N = [\lambda xy. \text{AND}(1 \cdot \text{iff}_2^B(x, y))]$$

In a similar way, $x <^N y$ if there is a bit position where $x$ has a 0, $y$ has a 1, and the portions of $x$ and $y$ to the left of that position are numerically equal.

**Definition 3.2.22**

$$<^N = [\lambda xy. \text{OR}(\text{rCRN}_2[\lambda xy. (x' <^B y') \land^B (x <^N y <))(lp_0(x, y), lp_0(y, x)))]$$

(Where we used “$i <^B j$” as shorthand for “$\lnot^B i \land^B j$”.)

Now, we prove basic properties of the two predicates just defined. A simple NIND suffices to show the following theorem.

**Theorem 3.2.13**

1. $x =^N \varepsilon \iff x = 0x$
2. $x =^N y \iff \text{lp}_0(x, y) = \text{lp}_0(y, x)$

This immediately implies that “$=^N$” is an equivalence relation. If we define $=^L = [\lambda xy. (x \triangleright y) \cdot (x < y)]^{\text{ZL}}(\varepsilon, 1)$ and $=^S = [\lambda xy. \lnot^B(x =^L y) \land^B x =^N y]$, then the theorem we just proved immediately implies that $x =^S y \iff x = y$. Together with the facts about propositional connectives proved in Theorem 3.2.6, this means that for any formula $A$ of $T_1$, there exists a term $\hat{A}$ of $T_1$ such that $T_1$ can prove $A \leftrightarrow \hat{A}$ (with our usual convention whereby “$\hat{A}$” stands for the formula $\hat{A} = 1$).

Now, we prove more properties of $=^N$ and $<^N$. By Theorem 3.2.10, we have the following two theorems.

**Claim 3.2.67**

1. $x0 =^N y0 \iff x =^N y \iff x1 =^N y1$
2. $\lnot^B(x0 =^N y1)$
3. \( \neg B(x1 =^N y0) \)

**Claim 3.2.68**

1. \( x0 <^N y0 = x <^N y \lor^B (x =^N y \land^B 0 <^B 0) = x <^N y \)
2. \( x0 <^N y1 = x <^N y \lor^B (x =^N y \land^B 0 <^B 1) = x \leq^N y \)
3. \( x1 <^N y0 = x <^N y \lor^B (x =^N y \land^B 1 <^B 0) = x <^N y \)
4. \( x1 <^N y1 = x <^N y \lor^B (x =^N y \land^B 1 <^B 1) = x <^N y \)

Simple proofs by NIND now suffice to show the following lemma.

**Claim 3.2.69**

1. \( \neg B(x <^N \varepsilon) \)
2. \( \neg B(x <^N x) \)
3. \( \neg B(\varepsilon <^N 0x) \)

Using the notation “\( x >^N y \)” for \( y <^N x \), “\( x \leq^N y \)” for \( x <^N y \lor^B x =^N y \), and “\( \geq^N \)” similarly defined, we have the following theorem. (We give its proof here because it is representative of the kind of proof that will be used for most theorems concerning numerical functions.)

**Theorem 3.2.14**

\[
(x <^N y \land^B \neg^B (x =^N y) \land^B \neg^B (x >^N y)) \\
\lor^B (\neg^B (x <^N y) \land^B x =^N y \land^B \neg^B (x >^N y)) \\
\lor^B (\neg^B (x <^N y) \land^B \neg^B (x =^N y) \land^B x >^N y)
\]

**Proof** By Derived Rule [3.2.3] and the lemma above: When \( y = \varepsilon \), the statement of the theorem reduces to \( (x =^N \varepsilon \land^B \neg^B (x >^N \varepsilon)) \lor^B (\neg^B (x =^N \varepsilon) \land^B x >^N \varepsilon) \), which can be proved by regular NIND on \( x \): \( \varepsilon =^N \varepsilon \land^B \neg^B (\varepsilon >^N \varepsilon), (x0 =^N \varepsilon \land^B \neg^B (x0 >^N \varepsilon)) \lor^B (\neg^B (x0 =^N \varepsilon) \land^B x0 >^N \varepsilon) = (x =^N \varepsilon \land^B \neg^B (x >^N \varepsilon)) \lor^B (\neg^B (x =^N \varepsilon) \land^B x >^N \varepsilon), (x1 =^N \varepsilon \land^B \neg^B (x1 >^N \varepsilon)) \lor^B (\neg^B (x1 =^N \varepsilon) \land^B x1 >^N \varepsilon) = \varepsilon <^N x \lor^B \varepsilon =^N x = \varepsilon \leq^N x \). We can show that the statement holds when \( x = \varepsilon \) in the same way. Next, we have four cases to consider:

\[
(x0 <^N y0 \land^B \neg^B (x0 =^N y0) \land^B \neg^B (x0 >^N y0)) \\
\lor^B (\neg^B (x0 <^N y0) \land^B x0 =^N y0 \land^B \neg^B (x0 >^N y0)) \\
\lor^B (\neg^B (x0 <^N y0) \land^B \neg^B (x0 =^N y0) \land^B x0 >^N y0) = (x <^N y \land^B \neg^B (x =^N y) \land^B \neg^B (x >^N y)) \\
\lor^B (\neg^B (x <^N y) \land^B x =^N y \land^B \neg^B (x >^N y)) \\
\lor^B (\neg^B (x <^N y) \land^B \neg^B (x =^N y) \land^B x >^N y)
\]
and similarly for $x_1, y_1$,

$$(x_0 <^N y_1 \land B (x_0 =^N y_1) \land B (x_0 >^N y_1))$$

$$\lor^B (B (x_0 <^N y_1) \land B x_0 =^N y_1 \land B (x_0 >^N y_1))$$

$$\lor^B (\neg B (x_0 <^N y_1) \land B x_0 =^N y_1 \land B x_0 >^N y_1) = (x \leq^N y \land B x =^N y \land B (x >^N y))$$

$$\lor^B (\neg B (x \leq^N y) \land B 0 \land B \neg B (x >^N y))$$

$$\lor^B (\neg B (x \leq^N y) \land B 1 \land B x >^N y)$$

$$= (x <^N y \land B \neg B (x >^N y))$$

$$\lor^B (x =^N y \land B \neg B (x >^N y))$$

$$\lor^B (\neg B (x <^N y) \land B \neg B (x =^N y) \land B x >^N y)$$

and similarly for $x_1, y_0$. □

**Corollary 3.2.70**  $\varepsilon \leq^N x$

**Corollary 3.2.71**  $x =^N y = x \leq^N y \land B x \geq^N y$

Next, from the fact that $l p_0 (0 x, y) = l p_0 (x, y) \lor l p_0 (0 x, y) = 0 \cdot l p_0 (x, y)$ (which can easily be proved by cases depending on the length of $x \parallel y$), simple proofs by Derived Rule [3.2.3] show the following lemma.

**Lemma 3.2.72**

1. $x =^N y = 0 x =^N 0 y = 0 x =^N 0 y$
2. $x <^N y = 0 x <^N 0 y$
3. $x <^N y = 0 x <^N y = x <^N 0 y$

This lemma can be used, with a generalization of Derived Rule [3.2.3] to three variables, to show the following theorems and their corollaries (from Theorem 3.2.13).

**Theorem 3.2.15**  $x =^N y \land y <^N z \rightarrow x <^N z$ and $x =^N y \land y >^N z \rightarrow x >^N z$

**Corollary 3.2.73**  $x =^N y \land y \leq^N z \rightarrow x \leq^N z$ and $x =^N y \land y \geq^N z \rightarrow x \geq^N z$

**Theorem 3.2.16**  $x <^N y \land y <^N z \rightarrow x <^N z$ and $x >^N y \land y >^N z \rightarrow x >^N z$

**Corollary 3.2.74**  $x \leq^N y \land y <^N z \rightarrow x <^N z$ and $x \geq^N y \land y >^N z \rightarrow x >^N z$

**Corollary 3.2.75**  $x \leq^N y \land y \leq^N z \rightarrow x \leq^N z$ and $x \geq^N y \land y \geq^N z \rightarrow x \geq^N z$
On “|·|” and “succN”

Now, we define the binary length function “|·|” and the numerical successor function “succN” as in $L_1$, and prove some of their basic properties.

**Definition 3.2.23**

\[
cussN = \ell \text{CRN} \left[ \lambda x . \text{AND}(1 \succ x) ?^B (-\ell^a x, \ell x) \right] \\
\text{succN} = [\lambda x . \text{cussN}(0x)]
\]

Simple proofs by NIND show the following theorem (proving the relevant properties first for the auxiliary function cussN, and then for succN).

**Claim 3.2.76**

\[
\text{succN}(\varepsilon) = 1 \\
\text{succN}(x0) = 0x1 \\
\text{succN}(x1) = \text{succN}(x) \cdot 0
\]

Using this theorem, a simple NIND will now prove the following properties.

**Claim 3.2.77**

\[
\text{succN}(0x) = 0 \cdot \text{succN}(x) \\
\text{succN}(1x) = \text{succN}(x) \cdot -^B \text{succN}(x) \cdot \succ N(x) \\
\text{succN}(x) = \text{AND}(1x) \\
\text{succN}(x) = \text{AND}(1x) ?^B (1 \cdot 0x, 0 \cdot \succ N(x))
\]

Now, we can prove a few theorems involving succN together with some of the other numerical functions already defined.

**Theorem 3.2.17**

1. $x <^N \text{succN}(x)$
2. $x >^N y = x \geq^N \text{succN}(y)$

The binary length function is defined in the same way as in $L_1$, as follows.

**Definition 3.2.24**

\[
|·| = \text{STRN}[1, \lambda x v_\ell v_r . x ?^E \ell (v_\ell \cdot 0, v_\ell \cdot 1)]
\]

(To be consistent with previous notation, we will write “|x|” instead of the more formal “||x||”.)
3.2. Developing the theory

Claim 3.2.78  \[ |x| = |jx| \]

Theorem 3.2.18  \[ jx = jy \leftrightarrow |x| = |y| \]

Proof  One direction (\( jx = jy \rightarrow |x| = |y| \)) is immediate from the preceding claim. The other is proved by TIND on \( y \) (with \( h_\ell = \uparrow \) and \( h_r = \downarrow \)): \( |x| = |\varepsilon| \rightarrow |x| = \varepsilon \rightarrow x = \varepsilon \rightarrow jx = j\varepsilon, |x| = |i| \rightarrow |x| = 1 \rightarrow x = i' \rightarrow jx = j, \) and assuming that \( |x\uparrow| = |y\uparrow| \rightarrow jx\uparrow = jy\uparrow \) and \( |\uparrow x| = |\uparrow y| \rightarrow j\uparrow x = j\uparrow y, \) we have that

\[
|x| = |y| \rightarrow x \uparrow = y \uparrow \rightarrow (|x\downarrow| \cdot 0, |x\uparrow| \cdot 1) = y \uparrow = (|y\downarrow| \cdot 0, |y\uparrow| \cdot 1) \\
\rightarrow (x\uparrow \uparrow x = \varepsilon \land y\uparrow \uparrow y = \varepsilon \land |x\downarrow| \cdot 0 = |y\downarrow| \cdot 0) \lor \\
(x\uparrow \uparrow x \neq \varepsilon \land y\uparrow \uparrow y \neq \varepsilon \land |x\downarrow| \cdot 1 = |y\downarrow| \cdot 1) \\
\rightarrow (x\uparrow \uparrow x = \varepsilon \land y\uparrow \uparrow y = \varepsilon \land jx\uparrow = jy\uparrow) \lor \\
(x\uparrow \uparrow x = i \land y\uparrow \uparrow y = i' \land jx\uparrow = jy\uparrow) \\
\rightarrow jx = jy
\]

(where the two cases for \( |x\downarrow| \cdot 0 = |y\downarrow| \cdot 1 \) and \( |x\downarrow| \cdot 1 = |y\downarrow| \cdot 0 \) were not included in the disjunction on the second and third lines since they are known to be false). \( \Box \)

The following theorem can be proved with an easy TIND and its corollaries are immediate from previously proved theorems.

Theorem 3.2.19  \[ |xi| =^N \text{succ}^N(|x|) \]

Corollary 3.2.79  \[ |x| <^N |xi| \]

Corollary 3.2.80  \[ x >^N |y| \rightarrow x \geq^N |y| \]

On “masking” functions

In order to define binary addition, and to prove its properties, we will need “masking” functions like the ones that were defined in \( L_1 \). We give their definition and basic properties here.

Definition 3.2.25  
\[
\text{first}_0 = r\text{CRN}[\lambda x. \text{AND}(1x^<) \uparrow (\overline{0}x^', 0)] \\
\text{first}_1 = r\text{CRN}[\lambda x. \text{OR}(x^<) \uparrow (0, x^')] 
\]

Definition 3.2.26  \[ \text{maskbit} = [\lambda xy. \text{OR}(\text{and}_2^B(x, y))] \]

Definition 3.2.27  \[ \text{delfirst}_1 = [\lambda x. \text{and}_2^B(x, \overline{\text{not}^B(\text{first}_1(x))})] \]
The basic theorem below, as well as its corollary, can both be proved with a simple NIND.

**Theorem 3.2.20**

\[
\begin{align*}
\text{first}_0(0x) &= 1 \cdot 0x \\
\text{first}_0(1x) &= 0 \cdot \text{first}_0(x) \\
\text{first}_1(0x) &= 0 \cdot \text{first}_1(x) \\
\text{first}_1(1x) &= 1 \cdot 0x
\end{align*}
\]

**Corollary 3.2.81** \(\text{first}_0(x) = \text{first}_1(\text{not}^N(x))\)

**On binary addition**

Before we define binary addition and prove its properties, let us make a remark about “numerical” functions. If a formula \(A\) contains only terms made up of functions \(f\) with the property that \(f(x_1, \ldots, x_m) = f(l_0(x_1, \max^m_0(\overline{x}_m)), \ldots, l_0(x_m, \max^m_0(\overline{x}_m)))\) (which happens to be the case for the numerical functions), then \(A[x_1, \ldots, x_m] \leftrightarrow (jx_1 = \cdots = jx_m \rightarrow A[x_1, \ldots, x_m])\).

Thus, we can use the following special form of Derived Rule 3.2.3 to prove any such formula \(A\) (the rule is stated only for two variables but can easily be extended to more).

### Derived Rule 3.2.6

\[
A[\varepsilon, \varepsilon], jx = jy \land A[x, y] \rightarrow A[0x, 0y] \land A[0x, 1y] \land A[1x, 0y] \land A[1x, 1y]
\]

\(\vdash jx = jy \rightarrow A[x, y]\)

(The conclusion of the rule can easily be proved from the antecedent by a simple application of Derived Rule 3.2.3.)

Now, binary addition is defined just as in \(L_1\), as follows.

**Definition 3.2.28** \(\text{carry}^N = \ell\text{CRN}_2[\lambda xy. \text{maskbit}(\text{and}_2^N(x, y), \text{first}_0(\text{xor}_2^N(x, y)))]\)

**Definition 3.2.29** \(+^N = [\lambda xy. \text{xor}_3^N(\text{carry}^N(x, y) \cdot 0, x, y)]\)

(To make the notation consistent with previous usage, we will write “\(x +^N y\)” instead of the more formal “\(+^N(x, y)\)”.) The commutativity of “\(+^N\)” is a direct result of the commutativity of each function involved in its definition.

**Theorem 3.2.21** \(x +^N y = y +^N x\)

Proving the associativity of “\(+^N\)” will be slightly more complicated. First, we relate the functions \(+^N\) and \(\text{succ}^N\) through the following lemma and theorem.

**Lemma 3.2.82**

\[
\begin{align*}
\text{carry}^N(x0, 1) &= \text{carry}^N(x, \varepsilon) \cdot 0 = 0x0 \\
\text{carry}^N(x1, 1) &= x ?^2z (1, \text{carry}^N(x, 1) \cdot 1)
\end{align*}
\]
3.2. Developing the theory

**Theorem 3.2.22** \( x +^N 1 = x \oplus (0 \cdot \text{succ}^N(x), \text{succ}^N(x)) =^N \text{succ}^N(x) \)

Next, we can state certain facts about the carry function.

**Claim 3.2.83** For \( jx = jy \),

\[
\begin{align*}
\text{carry}^N(x, \varepsilon) &= 0x \\
\text{carry}^N(x, x) &= x \\
\text{carry}^N(0x, 0y) &= 0 \cdot \text{carry}^N(x, y) \\
\text{carry}^N(1x, 0y) &= \text{carry}^N(x, y) \cdot \text{carry}^N(x, y) \\
\text{carry}^N(1x, 1y) &= 1 \cdot \text{carry}^N(x, y)
\end{align*}
\]

Note that we omitted the property \( \text{carry}^N(0x, 1y) = \text{carry}^N(x, y) \cdot \text{carry}^N(x, y) \) from this theorem since it follows directly by the commutativity of \( \text{carry}^N \). This will be the case for many of the theorems and proofs about \( +^N \) that we will now present: for the sake of brevity, we will omit statements and proofs that follow directly from previous ones by commutativity. The following claim follows directly from the corresponding properties for \( \text{carry}^N \).

**Claim 3.2.84** For \( jx = jy \),

\[
\begin{align*}
x +^N \varepsilon &= 0x \\
x +^N x &= x0 \\
0x +^N 0y &= 0 \cdot (x +^N y) \\
1x +^N 0y &= \langle x +^N y \rangle \cdot \text{succ}^N(x +^N y) \cdot \text{succ}^N(x +^N y) \\
1x +^N 1y &= 1 \cdot (x +^N y)
\end{align*}
\]

Now, although we can use Claim 3.2.84 to prove theorems about \( +^N \) by Derived Rule 3.2.6, we will also have need of the following theorem further on.

**Claim 3.2.85**

\[
\begin{align*}
x0 +^N y0 &= (x +^N y) \cdot 0 \\
x1 +^N y0 &= (x +^N y) \cdot 1 \\
x1 +^N y1 &= \text{succ}^N(x +^N y) \cdot 0
\end{align*}
\]

With the help of this theorem, we can now prove the following important properties of \( +^N \) with a version of Derived Rule 3.2.6 that concatenates bits to the right instead of to the left.
Theorem 3.2.23

1. \( x +^N \text{succ}^N(y) =^N \text{succ}^N(x +^N y) \)
2. \( x +^N (y +^N z) =^N (x +^N y) +^N z \)
3. \( y <^N z \iff x +^N y <^N x +^N z \)
4. \( x =^N y \land z =^N w \rightarrow x +^N z =^N y +^N w \)
5. \( x =^N y \land z <^N w \rightarrow x +^N z <^N y +^N w \)
6. \( x <^N y \land z <^N w \rightarrow x +^N z <^N y +^N w \)

On iterated sums

The last functions we need to define are iterated sums, defined as in \( L_1 \) using Buss’s “carry-save” technique.

Definition 3.2.30

\[
\begin{align*}
\text{CScar}_3 &= \ell\text{CRN}_3[\lambda x_1 x_2 x_3. (\langle x_1 \land^B x_2 \rangle \lor^B (\langle x_2 \land^B x_3 \rangle \lor^B \langle x_3 \land^B x_1 \rangle))] \\
\text{CSadd}_3 &= [\lambda x_1 x_2 x_3. \text{xor}_3^B(0x_1, 0x_2, 0x_3)] \\
\text{CScar} &= [\lambda x_1 x_2 x_3 x_4. \text{CScar}_3(\text{CScar}_3(x_1, x_2, x_3) \cdot 0, \text{CSadd}_3(x_1, x_2, x_3), 0x_4) \cdot 0] \\
\text{CSadd} &= [\lambda x_1 x_2 x_3 x_4. \text{CSadd}_3(\text{CScar}_3(x_1, x_2, x_3) \cdot 0, \text{CSadd}_3(x_1, x_2, x_3), 0x_4)]
\end{align*}
\]

The following properties are a direct consequence of these definitions.

Claim 3.2.86

\[
\begin{align*}
\text{CScar}_3(x_0, y_0, 0, 0) &= \text{CScar}_3(x, y, z) \cdot 0 \\
\text{CScar}_3(x_1, y_0, 0, 0) &= \text{CScar}_3(x_0, y_1, 0, 0) = \text{CScar}_3(x_1, y_0, 0, 1) = \text{CScar}_3(x, y, z) \cdot 0 \\
\text{CScar}_3(x_0, y_1, 0, 0) &= \text{CScar}_3(x_1, y_0, 0, 1) = \text{CScar}_3(x_1, y_1, 0, 0) = \text{CScar}_3(x, y, z) \cdot 1 \\
\text{CScar}_3(x_1, y_1, 1, 1) &= \text{CScar}_3(x, y, z) \cdot 1 \\
0 \cdot \text{CScar}_3(x, y, z) &= 0 \cdot 0 \text{max}_3^2(x, y, z) = 0 \cdot \text{max}_3^2(0x, 0y, 0z) \\
0 \cdot \text{max}_3^2(0x, 0y, 0z) &= \text{CScar}_3(0x, 0y, 0z) \cdot 0 = \text{CSadd}_3(0x, 0y, 0z) \\
0 \cdot \text{max}_3^2(0x, 0y, 0z, 0w) &= 0 \cdot 0 \text{max}_4^3(x, y, z, w) = 00 \cdot \text{max}_4^3(0x, 0y, 0z, 0w) \\
00 \cdot \text{max}_4^3(0x, 0y, 0z, 0w) &= \text{CScar}(0x, 0y, 0z, 0w) = \text{CSadd}(0x, 0y, 0z, 0w)
\end{align*}
\]

We can now prove one main lemma and one main theorem about the “carry-save” addition functions.
3.3 Proving the pigeonhole principle in $T_1$

In this section, we will be working with the following form of the pigeonhole principle, denoted PHP$_n(f)$ (or simply PHP when $n$ and $f$ are clear from the context): “no map $f : [n+1] \to [n]$ is injective”, or equivalently “if $f$ is a map from $[n+1]$ to $[n]$, then there exist $i \neq j \in [n+1]$ such that $f(i) = f(j)$”. (Note that we are using the common notation “[n]” to represent the
set \{1, 2, \ldots, n\}, for any positive integer \(n\), and in what follows, we will use the term \textit{map} to mean a (possibly) multi-valued function.

Informally, the proof of PHP goes as follows: Assume for a contradiction that \(f\) is a map from \([n+1]\) to \([n]\) and that \(f\) is injective (\(i.e., \) for every \(i \neq j \in [n+1] , f(i) \neq f(j)\)). Define
\[
\text{count}(k, \ell) = |\{x \in [\ell] : f(y) = x \text{ for some } y \in [k]\}|,
\]
i.e., \(\text{count}(k, \ell)\) is the number of elements in \([\ell]\) mapped onto from elements in \([k]\) by \(f\). Then, the following facts are easy to prove.

1. \(\text{count}(n+1, \ell) \leq \ell\) for any \(1 \leq \ell \leq n\) (since there are \(\ell\) elements in \([\ell]\)).
2. \(\text{count}(1, n) \geq 1\) (since \(f(1) \in [n]\)).
3. \(\text{count}(k+1, n) > \text{count}(k, n)\) for \(1 \leq k \leq n\) (since \(f(k+1)\) must be different from \(f(1), \ldots, f(k)\) by the assumption that \(f\) is injective).

Combining facts 2 and 3, we get that \(\text{count}(k, n) \geq k\) for all \(1 \leq k \leq n+1\). But then, \(n+1 \leq \text{count}(n+1, n) \leq n\), \(i.e., \) \(n+1 \leq n\), which is a contradiction. Hence, PHP is true.

In the rest of this section, we will show how the informal proof given above can be formalized in \(T_1\), in a top-down manner. Also, we adopt the following notational convention: when a function is defined through an auxiliary function that is of no interest in itself, the name of the auxiliary function will consist of the function’s name spelled backwards (\(e.g.,\) we will define below a function \textit{“map”} in terms of an auxiliary function \textit{“pam”}).

### 3.3.1 Representation of PHP in \(T_1\)

The first step in the formalization will be to use the machinery given above to write down \(L_1\)-functions that define PHP. Formally, \(\text{PHP} = \text{PHP}_n(f)\) depends on two parameters: \(n\) and \(f\); moreover, given \(n, f\) can be described by an \([n \times (n+1)]\) binary array whose \((i, j)\)-th entry is equal to 1 if \(i = f(j)\) and 0 otherwise, \(i.e.,\) each row corresponds to one hole and each column to one pigeon. This is essentially the representation we will use to encode the problem.

More precisely, given \(n\), every bit string \(a\) can be seen as encoding an \([n \times (n+1)]\) binary array (and therefore a partial map \(f^a_n : [n+1] \rightarrow [n]\)) by either padding \(a\) on the right with 0’s or chopping off enough bits from the right of \(a\) so that its length is \(n \cdot (n+1)\), and then reading the array in row-major order, so that the first \(n+1\) bits of \(a\) (from the left) represent the first row of the array, and so on. For example, the string 100110010 represents at the same time

\[
\text{the } [2 \times 3] \text{ binary array } \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}, \quad \text{the } [3 \times 4] \text{ binary array } \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \quad \text{etc.}
\]
3.3. Proving the pigeonhole principle in $T_1$

Then, every such $[n \times (n + 1)]$ binary array $a$ represents a partial map $f^n_\alpha : [n + 1] \to [n]$.

Now, given a bit string $N$ such that $n = |N|$, we can define a function $\text{adj}(a, N)$ that “adjusts” the length of the bit string $a$ so that $|\text{adj}(a, N)| = |N| \cdot (|N| + 1)$, i.e., $\text{adj}(a, N)$ is exactly the $[n \times (n + 1)]$ binary matrix encoded by $a$, written out in row-major order:

$$\text{adj}(a, N) = a \triangleleft ((N1 \# N) \triangleright a) \cdot 0(a \triangleright (N1 \# N)).$$

Next, given a column number $k$ and a row number $l$ in unary, we can easily define a function $\text{entry}$ that extracts a single entry (bit) of the matrix:

$$\text{entry}(a, n, k, l) = ((N1 \# L\triangleleft \cdot K\triangleleft) \triangleright \text{adj}(a, N))'.$$

Note that the value returned by this function is meaningless unless $1 \leq |k| = k \leq n + 1$ and $1 \leq |l| = \ell \leq n$. In order to simplify the presentation, we will implicitly assume that $k$ and $\ell$ fall within this range for the rest of the section, where $k = |K|$ and $\ell = |L|$ by convention (i.e., functions are implicitly defined by cases to be equal to $\varepsilon$ for values outside the meaningful range).

Once we have the function $\text{entry}$, it is easy to define functions $\text{col}$ and $\text{row}$ that extract columns or rows of the matrix, by CRN:

$$\text{loc}(a, n, k, l) = \text{loc}(a, n, k, L\triangleleft \cdot \text{entry}(a, n, k, L)) \quad (\text{for } L \neq \varepsilon)$$

$$\text{col}(a, n, K) = \text{loc}(a, n, K, N);$$

$$\text{wor}(a, n, K, L) = \text{wor}(a, n, K\triangleleft, L \cdot \text{entry}(a, n, K, L)) \quad (\text{for } K \neq \varepsilon)$$

$$\text{row}(a, n, L) = \text{wor}(a, n, N1, L).$$

So that $\text{col}(a, n, K)$ is the $k$-th column of $a$ and $\text{row}(a, n, L)$ is the $\ell$-th row of $a$. Moreover, it is easy to prove in $T_1$ that

$$\text{lc}(\text{col}(a, n, K), L) = \text{loc}(a, n, K, L),$$

$$\text{lc}(\text{row}(a, n, L), K) = \text{wor}(a, n, K, L),$$

directly from the properties of CRN.

Using these functions, we can now define two functions needed to represent PHP$_n(f)$:

$$\text{map}(a, N) = \begin{cases} 1 & \text{if } f^n_\alpha \text{ is a map, i.e., every column of } a \text{ contains at least one } 1, \\ 0 & \text{otherwise}; \end{cases}$$

$$\text{inj}(a, N) = \begin{cases} 1 & \text{if } f^n_\alpha \text{ is injective, i.e., every row of } a \text{ contains at most one } 1, \\ 0 & \text{otherwise}. \end{cases}$$
To compute \textbf{map} (resp. \textbf{inj}), we will first define a function \textbf{pam} (resp. \textbf{jni}) that returns a bit string with one bit for each column (resp. row) of \( a \) indicating whether the constraint is satisfied or not for that column (resp. row); then, we simply take the conjunction of all the bits to get the answer:

\[
\text{pam}(a, N, x) = \text{pam}(a, N, x^\epsilon) \cdot \text{OR} \left( \text{col}(a, N, x) \right) \quad (\text{for } x \neq \epsilon)
\]

\[
\text{map}(a, N) = \text{AND} \left( \text{pam}(a, N, N1) \right),
\]

\[
\text{jni}(a, N, x) = \text{jni}(a, N, x^\epsilon) \cdot \neg^B \text{OR} \left( \text{delfirst}_1 \left( \text{row}(a, N, x) \right) \right) \quad (\text{for } x \neq \epsilon)
\]

\[
\text{inj}(a, N) = \text{AND} \left( \text{jni}(a, N, N) \right).
\]

Finally, we can easily define a function that represents PHP:

\[
\text{php}(a, N) = \text{map}(a, N) \rightarrow^B \neg^B \text{inj}(a, N).
\]

### 3.3.2 The \( T_1 \)-proof of PHP

First, let us define the function \textbf{count}(a, N, K, L), which returns the number of elements from \([\ell]\) that are mapped onto by elements from \([k]\) according to \( f^n_{\ell} \). We do this by first defining \textbf{tnuoc}(a, N, K, L), which returns a string of \( \ell \) bits, one for each of the first \( \ell \) rows of \( a \), where bit \( j \) is set to 1 iff row \( j \) contains at least one 1 in the first \( k \) columns:

\[
\text{tnuoc}(a, N, K, L) = \text{tnuoc}(a, N, K, L^\epsilon) \cdot \text{OR} \left( \text{wor}(a, N, K, L) \right) \quad (\text{for } L \neq \epsilon)
\]

Then, \( \text{count}(a, N, K, L) = \text{sum}(\text{tnuoc}(a, N, K, L)) \). We can depict the situation as follows, where we have represented the submatrix of \( a \) consisting of the first \( k \) columns for each of the first \( \ell \) rows, and where the value of \( \text{tnuoc}(a, N, K, L) \) can be read bit by bit, one for each row:

\[
\begin{align*}
\text{tnuoc}(a, N, K, L) : \\
\ell \quad &| \quad k \\
\begin{array}{cccccc}
1 & 0 & 0 & \cdots & 1 & \rightarrow & 1 = \text{OR}(\text{wor}(a, N, K, 1)) \\
0 & 0 & 0 & \cdots & 0 & \rightarrow & 0 = \text{OR}(\text{wor}(a, N, K, 11)) \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & 1 & \rightarrow & 1 = \text{OR}(\text{wor}(a, N, K, L))
\end{array}
\end{align*}
\]

Now, we give a formalization in \( T_1 \) of the high-level proof of PHP outlined at the beginning of this section. To make the notation easier to read, all theorems are conditional to the fact that \( k \) and \( \ell \) are within meaningful range.

Recall the general outline of the proof: under the assumption that \( \neg^B \text{php}(a, N) \), \textit{i.e.}, that \( \text{map}(a, N) \) and \( \text{inj}(a, N) \), it is possible to prove the following two facts.

\textbf{FACT 3.3.1} \quad \text{count}(a, N, N1, L) \leq^N \ell \quad (\text{where } \ell = |L|, \text{by convention})
3.3. Proving the pigeonhole principle in $T_1$

**Fact 3.3.2** \( k \leq^N \text{count}(a, N, k, N) \) (where \( k = |K| \), by convention)

Then, we get that \( |N1| \leq^N \text{count}(a, N, N1, N) \leq^N |N| \), so that \( |N1| \leq^N |N| \) (by transitivity of \( \leq^N \)). But since we know that \( \neg^B (|x1| \leq^N |x|) \), we get \( \text{php}(a, N) \) by contradiction.

Now, we can prove fact **3.3.3**

\[
\text{count}(a, N, N1, L) = \sum(\text{tnuoc}(a, N, N1, L)) \leq^N |\text{tnuoc}(a, N, N1, L)| \leq^N |L| = \ell
\]

since \( \sum(x) \leq^N |x| \) for any string \( x \) and \( |r\text{CRN}[h](x, \vec{y})| = |x| \) for any \( L_1 \)-function \( h \) and any strings \( x, \vec{y} \).

To prove fact **3.3.2**, we will first show that it is possible to prove the following two facts (corresponding to facts \( \text{2} \) and \( \text{3} \) in the informal proof).

**Fact 3.3.3** \( \text{count}(a, N, \varepsilon, N) =^N 0 \)

**Fact 3.3.4** \( \text{count}(a, N, K, N) >^N \text{count}(a, N, K <, N) \)

Then, we can use \( \text{NIND} \) to show that \( \text{count}(a, N, K, N) \geq^N |K| : \text{count}(a, N, \varepsilon, N) \geq^N |\varepsilon| \) by fact **3.3.3** and \( \text{count}(a, N, K, N) >^N \text{count}(a, N, K <, N) \geq^N |K <| \) by fact **3.3.4** and the induction hypothesis, so that \( \text{count}(a, N, K, N) \geq^N |K| \).

Next, to prove fact **3.3.3**, we use \( \text{NIND} \) together with the fact that \( \text{wor}(a, N, \varepsilon, L) = \varepsilon \) (by definition) to conclude that \( \text{tnuoc}(a, N, \varepsilon, L) = 0_L : \text{tnuoc}(a, N, \varepsilon, \varepsilon) = \varepsilon = 0\varepsilon \), and assuming that \( \text{tnuoc}(a, N, \varepsilon, L <) = 0(L <) \), then \( \text{tnuoc}(a, N, \varepsilon, L) = \text{tnuoc}(a, N, \varepsilon, L <) \cdot \text{OR}(\text{wor}(a, N, \varepsilon, L)) = 0(L <) \cdot \text{OR}(\varepsilon) = 0(L <) \cdot 0 = 0(L) \). Finally, we use the fact that \( \sum(\varepsilon x) =^N 0 \) for any string \( x \) to conclude that fact **3.3.3** holds.

To prove fact **3.3.4**, we have to show that \( \sum(\text{tnuoc}(a, N, K, N)) >^N \sum(\text{tnuoc}(a, N, K <, N)) \).

Intuitively, this will be true iff \( \text{tnuoc}(a, N, K, N) \) contains more 1’s than \( \text{tnuoc}(a, N, K <, N) \). Formally, for any two strings \( x \) and \( y \), the term \( \text{AND}(\text{or}_2^B(\text{not}^B(x), y)) \) expresses the fact that \( y \) has a 1 in every position where \( x \) has a 1, and the term \( \text{OR}(\text{and}_2^B(\text{not}^B(x), y)) \) expresses the fact that there is a position where \( x \) has a 0 but \( y \) has a 1. Now, since we can prove that \( \text{AND}(\text{or}_2^B(\text{not}^B(x), y)) \land^B \text{OR}(\text{and}_2^B(\text{not}^B(x), y)) \rightarrow^B \sum(x) <^N \sum(y) \) (the proof is given in Appendix \( \text{A} \)), we only need to show the following facts to complete the proof of fact **3.3.4**

**Fact 3.3.5** \( \text{AND}(\text{or}_2^B(\text{not}^B(\text{tnuoc}(a, N, K <, N)), \text{tnuoc}(a, N, K, N))) \)

**Fact 3.3.6** \( \text{OR}(\text{and}_2^B(\text{not}^B(\text{tnuoc}(a, N, K <, N)), \text{tnuoc}(a, N, K, N))) \)

It is relatively easy to prove fact **3.3.5** by \( \text{NIND} \) on the last argument: \( \text{tnuoc}(a, N, K <, 1) = \)}
\[ \text{OR}(\text{wor}(a, N, K, 1)) \] and \( \text{tnuoc}(a, N, K, 1) = \text{OR}(\text{wor}(a, N, K, 1)) \) so

\[
\begin{align*}
\text{AND}(\text{or}_B^2(\neg(\text{tnuoc}(a, N, K, 1)), \text{tnuoc}(a, N, K, 1))) \\
= -B \text{OR}(\text{wor}(a, N, K, 1)) & \lor \text{OR}(\text{wor}(a, N, K, 1)) \\
= -B \text{OR}(\text{wor}(a, N, K, 1)) & \lor B \text{OR}(\text{wor}(a, N, K, 1)) \\
= -B \text{OR}(\text{wor}(a, N, K, 1)) & \lor B \text{OR}(\text{wor}(a, N, K, 1)) \\
= 1 & \lor B \text{entry}(a, N, K, 1) = 1.
\end{align*}
\]

Also, by the definition of \text{tnuoc}, we get that

\[
\begin{align*}
\text{AND}(\text{or}_B^2(\neg(\text{tnuoc}(a, N, K, 1)), \text{tnuoc}(a, N, K, 1))) \\
= \text{AND}(\text{or}_B^2(\neg(\text{tnuoc}(a, N, K, 1) \land \text{OR}(\text{wor}(a, N, K, 1)))), \\
\quad \text{tnuoc}(a, N, K, 1) \land \text{OR}(\text{wor}(a, N, K, 1))) \\
= \text{AND}(\text{or}_B^2(\neg(\text{tnuoc}(a, N, K, 1) \land \text{OR}(\text{wor}(a, N, K, 1)))), \\
\quad \text{tnuoc}(a, N, K, 1) \land \text{OR}(\text{wor}(a, N, K, 1))) \\
\land B (\neg B \text{OR}(\text{wor}(a, N, K, 1)) \lor B \text{OR}(\text{wor}(a, N, K, 1))) \\
= 1 \land B (\neg B \text{OR}(\text{wor}(a, N, K, 1)) \lor B \text{OR}(\text{wor}(a, N, K, 1))) = 1
\end{align*}
\]

(where the third equality holds by the induction hypothesis).

The proof of fact 3.3.6 is the most involved so far. First, by the definition of \text{tnuoc} and properties of CRN, we know that

\[ \text{tnuoc}(a, N, K, N) = \text{tnuoc}(a, N, K, L) \land \text{OR}(\text{wor}(a, N, K, L)) \cdot \left( L \triangleright \text{tnuoc}(a, N, K, N) \right). \]

Because \( \text{OR}(xy) = \text{OR}(x) \lor B \text{OR}(y) \) and \( \text{and}_B^2(xy, wz) = \text{and}_B^2(x, w) \cdot \text{and}_B^2(y, z) \) when \( |x| = |w| \) and \( |y| = |z| \), we get easy proofs in \( T_1 \) that

\[
\neg B \text{OR}(\text{wor}(a, N, K, L)) \land B \text{OR}(\text{wor}(a, N, K, L)) \rightarrow B \text{OR}(\text{and}_B^2(\neg B(\text{tnuoc}(a, N, K, N)), \text{tnuoc}(a, N, K, N))).
\]

Hence, we can prove fact 3.3.6 by showing that there must exist some value \( L \) for which \( \neg B \text{OR}(\text{wor}(a, N, K, L)) \land B \text{OR}(\text{wor}(a, N, K, L)) \). Now, we can prove that \( \neg B \text{OR}(\text{delfirst}_1(x)) \land B \text{lb}(x, y) \rightarrow B \text{OR}(\text{lc}(x, y)) \) (see Appendix A), and since \( \text{wor}(a, N, K, L) = \text{lc}(\text{row}(a, N, L), K) \) and \( \neg B \text{OR}(\text{delfirst}_1(\text{row}(a, N, L))) \) by the assumption that \( \text{inj}(a, N) \), we only need to show that there is some \( L \) for which \( \text{lb}(\text{row}(a, N, L), K) \), which is equivalent to showing that there is a value of \( L \) for which \( \text{entry}(a, N, K, L) \).

Unfortunately, we do not have quantifiers to reason with so to show the existence of \( L \), we have to construct it explicitly, \( i.e., \) to define a function \( \text{pos}(a, N, K) \) that gives the value of \( L \). Because all functions definable in \( T_1 \) are length-determined, \( \text{pos} \) will have to return
3.3. Proving the pigeonhole principle in $T_1$

a bitmask to the position of $L$, and this bitmask cannot be used directly with the current definition of row to prove what we need. So, we will define an alternate function $mrow$ whose last argument is a bitmask instead of a unary string and for which we can show $row(a, N, L) = mrow(a, N, (1 \cdot 0(L \triangleright N1)) \leq)$:

$$mwor(a, N, K, m) = mwor(a, N, K \leq, m) \cdot \text{maskbit}^N(\text{col}(a, N, K), m) \quad (\text{for } K \neq \varepsilon)$$

$$mrow(a, N, m) = mwor(a, N, N1, m)$$

With this definition, pos can easily be defined as $pos(a, N, K) = \text{first}_1(\text{col}(a, N, K))$. By the assumption that $\text{map}(a, N)$, we know that $\text{OR}(\text{col}(a, N, K))$ and since we can prove that $\text{OR}(x) \leftrightarrow \text{maskbit}^N(x, \text{first}_1(x))$ in $T_1$ (see Appendix A), we get that $\text{maskbit}^N(\text{col}(a, N, K), pos(a, N, K))$, which implies immediately that $\text{lb}(mrow(a, N, pos(a, N, K)), K)$.

Now, because row and mrow are both defined by CRN on the same parameter $K$, it is sufficient to show that $entry(a, N, K, L) = \text{maskbit}^N(\text{col}(a, N, K), (1 \cdot 0(L \triangleright N1)) \leq)$ in order to prove that $row(a, N, L) = mrow(a, N, (1 \cdot 0(L \triangleright N1)) \leq)$. And because we can prove in $T_1$ that $\text{lb}^B(x, y) = \text{maskbit}^N(x, (1 \cdot 0(y \triangleright z1)) \leq)$ (see Appendix A), this is equivalent to showing that $entry(a, N, K, L) = \text{lb}(\text{col}(a, N, K), L) = \text{loc}(a, N, K, L)'$, a fact which is immediate by the definition of loc.

Finally, we can redefine $\text{mtnuoc}$ using $mwor$ instead of $wor$, as follows:

$$\text{mtnuoc}(a, N, K, L) = \text{mtnuoc}(a, N, K, L \leq) \cdot \text{OR}(mwor(a, N, K, (1 \cdot 0(L \triangleright N1)) \leq)) \quad (\text{for } L \neq \varepsilon)$$

and using the reasoning given above, it is possible to prove that

$$\text{OR}\left(\text{and}^B(\text{not}^B(\text{mtnuoc}(a, N, K \leq, N)), \text{mtnuoc}(a, N, K, N))\right).$$

Moreover, because of the equivalence between $wor$ and $mwor$ given above, this implies the corresponding result for $\text{tnuoc}$, which completes the proof.

Remark 3.3.1 Based on the $T_1$-proof of the pigeonhole principle just given, it should be possible to prove other, similar combinatorial statements in $T_1$. One example is Tutte’s theorem, which states that a graph has no perfect matching iff it satisfies a certain simple form of decomposition. This would give an alternative proof that “perfect matching” tautologies have short $F$-proofs, and maybe provide a more precise estimate of the size of these proofs by the results of Chapter 4. (The “perfect matching” tautologies were first discussed in a paper by Impagliazzo, Pitassi, and Urquhart [21], where it was shown that they have polysize $F$-proofs—note that those proofs were non-uniform, unlike the proofs we would obtain through $T_1$.)
Chapter 4

Theorems of $T_1$ Have Polysize $\mathcal{F}$-proofs

For every term $t$ of $T_1$ with free variables $x_1, \ldots, x_k$, we define a length function $\text{len}_t(m_1, \ldots, m_k)$ that gives the exact length of $t$ as a function of the lengths of $x_1, \ldots, x_k$ (this function is well-defined because functions in $L_1$ are length-determined). Then, we define a family of propositional term formulas $\langle t \rangle^{\vec{m}}_1, \ldots, \langle t \rangle^{\vec{m}}_{\text{len}_t(\vec{m})}$ that describe the bits of $t$ in terms of the bits of $x_1, \ldots, x_k$ (where $\langle t \rangle^{\vec{m}}_1$ describes the leftmost bit of $t$), i.e., given any truth-value assignment to the atoms representing the bits of $x_1, \ldots, x_n$, the truth value of $\langle t \rangle^{\vec{m}}_i$ represents the correct value for bit number $i$ of term $t$. Finally, for any formula $A$ of $T_1$, we define a family of propositional translations $[\ lbrack A \ rbrack]^\vec{m}$, where $\vec{m}$ lists the lengths of all free variables in $A$, and show that there are short $\mathcal{F}$-proofs of $[\ lbrack A \ rbrack]^\vec{m}$ whenever $A$ is a theorem of $T_1$.

4.1 Length functions

The length functions are defined inductively as follows (where “sg” is the signum function).

\[
\begin{align*}
\text{len}_x(m) &= m \\
\text{len}_f(t_1, \ldots, t_k)(\vec{m}) &= \text{len}_f(x_1, \ldots, x_k)(\text{len}_{t_1}(\vec{m}_1), \ldots, \text{len}_{t_k}(\vec{m}_k)) \quad (\text{where } x_1, \ldots, x_k \text{ occur in none of } t_1, \ldots, t_k) \\
\text{len}_\varepsilon &= 0 \\
\text{len}_0 &= 1 \\
\text{len}_1 &= 1 \\
\text{len}_{0x}(m) &= m \\
\text{len}_{1x}(m) &= m \\
\text{len}_{\uparrow x}(m) &= \lfloor m/2 \rfloor \\
\text{len}_{\downarrow x}(m) &= \lceil m/2 \rceil \\
\text{len}_{\prec y}(m, n) &= m \div n \\
\text{len}_{\succ y}(n, m) &= m \div n \\
\end{align*}
\]

\footnote{Where $\vec{m}_i$ represents the lengths of the variables that occur in $t_i$.}
To every variable $x$ of $T_1$ are associated propositional atoms $\langle x \rangle_i^1, \ldots, \langle x \rangle_i^m$. For other terms of $T_1$, the term formulas are defined inductively as follows. (When subscript $i$ is used without specifying its range in the definition of $(t)_i^\vec{m}$, it is implicitly assumed that $1 \leq i \leq \text{len}_t(\vec{m})$.)

$$\langle f(t_1, \ldots, t_k)_i^\vec{m} = \langle f(x_1, \ldots, x_k)_i^\text{len}_t(\vec{m}_1) \ldots \text{len}_t(\vec{m}_k) \left[ (t_j)_i^\vec{m}_j \langle / \rangle (x_j)_i^\text{len}_t(\vec{m}_j) \right] \right]_{1 \leq j \leq k}$$

( where $x_1, \ldots, x_k$ occur in none of $t_1, \ldots, t_k$ )

$$\langle 0 \rangle_i^1 = \bot \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \Quad
4.3 Propositional translations

\[ \langle \text{TRN}[g, h, h, h](x, z, \vec{y}) \rangle_{i}^{m,p,\vec{n}} = \begin{cases} \langle g(x, z, \vec{y}) \rangle_{i}^{m,p,\vec{n}} & \text{if } m \leq 1 \\ h(x, z, \vec{y}, \text{TRN}[g, h, h, h \downarrow, h \uparrow](x, h_{\ell}(z), \vec{y})) & \text{if } 1 < m \end{cases} \]

Remark 4.2.1 Note that "?" is the only primitive function symbol that has non-trivial term formulas (because it is the only function that depends directly on the values of its arguments), so that any non-trivial term formula must depend on ? in some way.

4.3 Propositional translations

The propositional translations of formulas of \( T_1 \) are defined inductively as follows (where \( \odot \) stands for any one of the binary propositional connectives, and \( \vec{m}_0 \) and \( \vec{m}_1 \) represent the lengths of the variables that occur in \( A \) and \( B \) (or \( t \) and \( u \)), respectively.)

\[
[ t = u] = \begin{cases} \bigwedge_{1 \leq i \leq \text{len}_t(\vec{m}_0)} \langle t \rangle_{i}^{\vec{m}_0} \leftrightarrow \langle u \rangle_{i}^{\vec{m}_1} & \text{if } \text{len}_t(\vec{m}_0) = \text{len}_u(\vec{m}_1), \\ \perp & \text{otherwise.} \end{cases}
\]

\[
[ \neg A] = \neg [A]
\]

\[
[A \odot B] = [A]^{\vec{m}_0} \odot [B]^{\vec{m}_1}
\]

4.4 The simulation result

Now, we can prove the following theorem.

Theorem 4.4.1 If \( A \) is provable in \( T_1 \), then for any \( \vec{m} \), \([A]^{\vec{m}}\) has uniform polysize \( \mathcal{F} \)-proofs.

Proof The proof is by induction on the number of inferences in the proof of \( A \). If \( A \) is an axiom, then Subsection 4.4.1 below shows that \([A]^{\vec{m}}\) has linear-size \( \mathcal{F} \)-proofs. If \( A \) is obtained by a derivation, then by the induction hypothesis, the propositional translations of the premises of the last inference all have short \( \mathcal{F} \)-proofs. Subsection 4.4.2 below shows that in this case also, \([A]^{\vec{m}}\) has short \( \mathcal{F} \)-proofs. Moreover, all these \( \mathcal{F} \)-proofs are uniform, in the sense that there exists a specific function that takes a theorem of \( T_1 \) and the lengths of its variables into a \( \mathcal{F} \)-proof of the translation of the theorem. This function is described implicitly in the sections that follow, but it can be formalized in \( T_1 \) itself, using techniques similar to those developed in Chapter 5 (we do not expect any technical difficulties in doing this but time constraints prevent us from working out the details). \( \square \)
4.4.1 Axioms

For most axioms of the form $t = u$, just writing down the definitions of $\langle t \rangle_i$ and $\langle u \rangle_i$ is enough to see that the axiom is a theorem with short proofs since $\langle t \rangle_i = \langle u \rangle_i$. We give a more detailed argument only for a few axioms.

0. The axioms for the propositional calculus can obviously be simulated by any $\mathcal{F}$-system.

1. (a) By reflexivity, there are linear-size $\mathcal{F}$-proofs of $\bigwedge \langle x \rangle_i^m \leftrightarrow \langle x \rangle_i^m$ for any variable $x$.

(b) If $m \neq n$, then the antecedent of the axiom translates to $\bot$ so the translation of the axiom is a trivial theorem. If $m = n$, then the commutativity of $\leftrightarrow$ gives linear-size $\mathcal{F}$-proofs of $(\bigwedge \langle x \rangle_i^m \leftrightarrow \langle y \rangle_i^n) \rightarrow (\bigwedge \langle y \rangle_i^n \leftrightarrow \langle x \rangle_i^m)$.

(c) If $m \neq n$ or $n \neq p$, then one of the antecedents of the axiom translates to $\bot$ so the translation of the axiom is a trivial theorem. If $m = n = p$, then the transitivity of $\leftrightarrow$ gives linear-size $\mathcal{F}$-proofs of $((\bigwedge \langle x \rangle_i^m \leftrightarrow \langle y \rangle_i^n) \land (\bigwedge \langle y \rangle_i^n \leftrightarrow \langle z \rangle_i^p)) \rightarrow (\bigwedge \langle x \rangle_i^m \leftrightarrow \langle z \rangle_i^p)$.

(d) An easy induction on the structure of the function symbol $f$, together with properties of $\leftrightarrow$, is sufficient to show that there are short $\mathcal{F}$-proofs of $(\bigwedge \langle x \rangle_i^{m_1} \leftrightarrow \langle y \rangle_i^{n_1} \land \cdots \land \bigwedge \langle k \rangle_i^{m_k} \leftrightarrow \langle k \rangle_i^{n_k}) \rightarrow \bigwedge (f(x_1, \ldots, x_k))_{i}^{m_1 \cdots m_k} \leftrightarrow \langle f(y_1, \ldots, y_k) \rangle_i^{n_1 \cdots n_k}$ when $m_1 = n_1, \ldots, m_k = n_k$ (the axiom’s translation becoming a trivial theorem otherwise as one of the antecedents translates to $\bot$). The size of these proofs is linear if $f$ does not contain any function defined by TRN; it is polynomial otherwise (since the size of the term formulas can be polynomial in the lengths of the variables).

2. $\langle 0 \rangle_1 = \bot$ and $\langle 1 \rangle_1 = \top$.

3. (a) For all $1 \leq i \leq m + n$, $\langle x \cdot \varepsilon \rangle_i^m = \langle x \rangle_i^m$, and for all $1 \leq i \leq m + n + 1$,

$$
\langle x \cdot 0 \rangle_i^{m,n} = \begin{cases} 
\langle x \rangle_i^m & \text{if } i \leq m \\
\langle y \cdot 0 \rangle_{i-m}^{n,n} & \text{if } i - m \leq n \lessgtr m < i \\
\langle 0 \rangle_{i-m-n}^{n,n} & \text{if } n < i - m
\end{cases}
$$

$$
\langle (x \cdot y) \cdot 0 \rangle_i^{m,n} = \begin{cases} 
\langle x \cdot y \rangle_i^{m,n} & \text{if } i \leq m \\
\langle x \rangle_i^m & \text{if } i \leq m + n \\
\langle y \rangle_{i-m}^n & \text{if } m < i \\
\langle 0 \rangle_{i-(m+n)}^n & \text{if } m + n < i
\end{cases}
$$

(similarly for $x \cdot y1 = (x \cdot y) \cdot 1$).

(b) $[x \cdot y = \varepsilon]^{m,n}$ holds iff $m + n = 0$, and $[x = \varepsilon \land y = \varepsilon]^{m,n}$ holds iff $m = 0$ and $n = 0$.

(c) $[x \cdot y = 0]^{m,n}$ holds iff $m + n = 1$ and $\langle x \rangle_1^1 = \bot$ or $\langle y \rangle_1^1 = \bot$. Similarly for $x \cdot y = 1$. 

4. (a) For $1 \leq i \leq m$, $\langle \epsilon \triangleright x \rangle_i^m = \langle x \rangle_i^m$ by definition; also, for $1 \leq i \leq m - (n + 1)$, 
\[ \langle (0y) \triangleright x \rangle_i^{m,n} = \langle x \rangle_{i+1}^{m,n} = \langle y \triangleright x \rangle_i^{m,n} = (0 \triangleright (y \triangleright x))_i^{m,n}, \] and the same reasoning applies to $\langle (1y) \triangleright x \rangle_i^{m,n}$.

(b) $\text{len}_{0 \triangleright \epsilon} = 0 - 1 = 0 = \text{len}_\epsilon$, and for $1 \leq i \leq m + 1 - 1$, $\langle 0 \triangleright (0x) \rangle_i^m = \langle 0x \rangle_i^m = \langle x \rangle_{i+1}^m$ (similarly for $\langle 0 \triangleright (1x) \rangle_i^m$). The same reasoning applies with “1 $\triangleright \ldots$” in place of “0 $\triangleright \ldots$”.

(c) $\text{len}_{y \triangleright x}(m, n) = m - n$ and $\text{len}_{x \triangleright y}(m, n) = \text{len}_{x \triangleright y_1}(m, n) = (n + 1) - m$, and $m - n = 0 \leftrightarrow m = n \leftrightarrow m < n + 1 \leftrightarrow 0 < (n + 1) - m \leftrightarrow (n + 1) - m \neq 0$.

5. (a) Similarly to 4a

(b) Similarly to 4b

(c) Similarly to 4c

6. (a) $[0 \epsilon = \epsilon] = \top$ since $\text{len}_{\epsilon} = \text{len}_\epsilon = 0$, and for $1 \leq i \leq m + 1$, $\langle 0(x0) \rangle_i^m = \langle 0(x1) \rangle_i^m = \bot$ and $\langle 0x \cdot 0 \rangle_i^m = \begin{cases} 
\langle 0x \rangle_i^m & \text{if } i \leq m, \\
\langle 0 \rangle_{i-m} & \text{if } m < i.
\end{cases}$

(b) Similarly to 6a

7. $\langle \epsilon \triangleright (x, y, z) \rangle_i^{m,n,p} = \langle x \rangle_i$; for $1 \leq i \leq m$, by definition; also, for $1 \leq i \leq \max\{n, p\}$,
\[ \langle (w0) \triangleright (x, y, z) \rangle_i^{k,m,n,p} = (\neg \langle w0 \rangle_k \land \langle 0(z < y) \cdot y \rangle_i^{n,p}) \lor (\langle w0 \rangle_k \land \langle 0(y < z) \cdot z \rangle_i^{n,p}) \]
\[ = (\neg \bot \land \langle 0(z < y) \cdot y \rangle_i^{n,p}) \lor (\bot \land \langle 0(y < z) \cdot z \rangle_i^{n,p}) \]
\[ \leftrightarrow \langle 0(z < y) \cdot y \rangle_i^{n,p}, \]
\[ \langle (w1) \triangleright (x, y, z) \rangle_i^{k,m,n,p} = (\neg \langle w1 \rangle_k \land \langle 0(z < y) \cdot y \rangle_i^{n,p}) \lor (\langle w1 \rangle_k \land \langle 0(y < z) \cdot z \rangle_i^{n,p}) \]
\[ = (\neg \top \land \langle 0(z < y) \cdot y \rangle_i^{n,p}) \lor (\top \land \langle 0(y < z) \cdot z \rangle_i^{n,p}) \]
\[ \leftrightarrow \langle 0(y < z) \cdot z \rangle_i^{n,p}. \]

8. (a) For all $1 \leq i \leq m$,
\[ \langle (x \downarrow) \cdot (\triangleleft x) \rangle_i^{m,n} = \begin{cases} 
\langle x \rangle_i^m & \text{if } i \leq \lfloor m/2 \rfloor \\
\langle x \rangle_i^{m, \left[ m/2 \right]} & \text{if } \lfloor m/2 \rfloor < i
\end{cases} \]

(b) $x \downarrow \triangleleft x = \epsilon$ since $\text{len}_{x \downarrow \triangleleft x}(m) = \text{len}_{x \downarrow \triangleleft x}(m) = \lfloor m/2 \rfloor < \lfloor m/2 \rfloor = 0$;
\[ 1 \triangleright (x \downarrow \triangleleft x) = \epsilon \text{ since } (\text{len}_{x \downarrow \triangleleft x}(m) = \text{len}_{x \downarrow \triangleleft x}(m)) \div 1 = \lfloor m/2 \rfloor \div \lfloor m/2 \rfloor - 1 = 0. \]

The last four axioms are easy to prove if we note the following two facts.
(a) For any term $t$, $t \triangleleft t = \varepsilon$ and therefore $\langle a(t \triangleleft t) \cdot t \rangle_i^m = \langle t \rangle_i^m$ for $1 \leq i \leq \text{len}_i(m)$.

(b) Within any $\mathcal{F}$-system, the identity $(-p \land q) \lor (p \land q) \leftrightarrow q$ has linear-size proofs.

Together, these facts show that

$$\langle x ? (y, z, z) \rangle_i^{m,n,p} \leftrightarrow \begin{cases} \langle y \rangle_i^n & \text{if } m = 0, \\ \langle z \rangle_i^p & \text{if } m > 0. \end{cases}$$

9. (a) Note that $\text{len}_{(x0)[]} (m) = \lceil (m + 1)/2 \rceil = \lfloor m/2 \rfloor$ and

$$\text{len}_{(x0)[]} (m) = \lceil (m + 1)/2 \rceil = \lfloor m/2 \rfloor$$

so both terms do have the same length.

Now, for $1 \leq i \leq \lceil m/2 \rceil$, $(x0)[]}_i^m = \langle x0 \rangle_i^m = \langle x \rangle_i^m$ and

$$\langle x \triangleright \triangleright x ? (x \triangleright x \cdot ((\triangleright x \cdot 0) \lt \triangleright x), x \triangleright ((\triangleright x \cdot 0) \lt \triangleright x)) \rangle_i^m$$

$$\leftrightarrow \begin{cases} \langle x \rangle_i^m & \text{if } \lceil m/2 \rceil = \lfloor m/2 \rfloor \\ \langle x \triangleright x \cdot ((\triangleright x \cdot 0) \lt \triangleright x) \rangle_i^m & \text{if } \lceil m/2 \rceil > \lfloor m/2 \rfloor \end{cases}$$

$$\leftrightarrow \begin{cases} \langle x \rangle_i^m & \text{if } \lceil m/2 \rceil = \lfloor m/2 \rfloor \\ \langle x \triangleright x \cdot ((\triangleright x \cdot 0) \lt \triangleright x) \rangle_i^m & \text{if } i \leq \lceil m/2 \rceil \text{ and } \lceil m/2 \rceil > \lfloor m/2 \rfloor \end{cases}$$

$$\leftrightarrow \begin{cases} \langle x \rangle_i^m & \text{if } \lceil m/2 \rceil = \lfloor m/2 \rfloor \\ \langle x \rangle_i^m & \text{if } i \leq \lceil m/2 \rceil \text{ and } \lceil m/2 \rceil > \lfloor m/2 \rfloor \end{cases}$$

$$\leftrightarrow \langle x \rangle_i^m$$

since $\langle ((\triangleright x \cdot 0) \lt \triangleright x) \rangle_i^m = \langle x \rangle_i^m_{\lceil m/2 \rceil}$. The case for $x1$ is identical.

(b) Note that $\text{len}_{(x0)[]} (m) = \lfloor (m + 1)/2 \rfloor = \lfloor m/2 \rfloor + 1$ and

$$\text{len}_{(x0)[]} (m) = \lfloor (m + 1)/2 \rfloor = \lfloor m/2 \rfloor + 1$$

so both terms do have the same length.
Now, for $1 \leq i \leq \lfloor m/2 \rfloor + 1$,

$$
\langle \vec{\bullet} (x0) \rangle_i^m = (x0)_{i+\lfloor (m+1)/2 \rfloor}^m = \begin{cases} 
\langle x \rangle_{i+\lfloor m/2 \rfloor}^m & \text{if } i \leq \lfloor m/2 \rfloor \\
(0)_1 & \text{if } i = \lfloor m/2 \rfloor + 1
\end{cases}
$$

and

$$
\langle \vec{x} \triangleright \triangleright x ? (\vec{\bullet} x \cdot 0, 1 \triangleright (\vec{\bullet} x ) \cdot 0), 1 \triangleright (\vec{\bullet} x \cdot 0) \rangle_i^m
\leftrightarrow \begin{cases} 
\langle \vec{\bullet} x \cdot 0 \rangle_i^m & \text{if } \lfloor m/2 \rfloor = \lfloor m/2 \rfloor \\
(1 \triangleright (\vec{\bullet} x \cdot 0))_i^m & \text{if } \lfloor m/2 \rfloor > \lfloor m/2 \rfloor
\end{cases}
\leftrightarrow \begin{cases} 
\langle \vec{x} \rangle_i^m & \text{if } i \leq \lfloor m/2 \rfloor \text{ and } \lfloor m/2 \rfloor = \lfloor m/2 \rfloor \\
(0)_1 & \text{if } i > \lfloor m/2 \rfloor
\end{cases}
\leftrightarrow \begin{cases} 
\langle \vec{x} \rangle_i^m & \text{if } i \leq \lfloor m/2 \rfloor \text{ and } \lfloor m/2 \rfloor = \lfloor m/2 \rfloor \\
(0)_1 & \text{if } i > \lfloor m/2 \rfloor
\end{cases}
\leftrightarrow \begin{cases} 
\langle x \rangle_i^m & \text{if } i \leq \lfloor m/2 \rfloor \\
(0)_1 & \text{if } i = \lfloor m/2 \rfloor + 1
\end{cases}
$$

The case for $x1$ is identical.

(c) Similarly to 9a.

(d) Similarly to 9b.

10. $\langle (\lambda \vec{x} . t) (\vec{x}) \rangle_i^m = \langle t [\vec{x} / \vec{x}] \rangle_i^{\vec{m'}} = \langle t \rangle_i^{\vec{m'}}$ for all $1 \leq i \leq \text{len}_t (\vec{m'})$.

11. (a) Since $\text{len}_{\ell \text{CRN}} [h] (\varepsilon, \vec{y}) (\vec{m}) = 0$, $\ell \text{CRN}[h] (\varepsilon, \vec{y}) = \varepsilon [\vec{m}] = \top$.

Also, because $\text{len}_{(z \cdot 0)\varepsilon_2} (p) = p + 1 \div p = 1$, we have that

$$
\langle \ell \text{CRN}[h] (0x, \vec{y}) \rangle_i^m = \langle h (z, \vec{y}) \cdot 0 \rangle_i^{m+1, \vec{m}} = \langle 0 \rangle_j^m \cdot \langle z \rangle_j^{m+1} = \langle h (0x, \vec{y}) \cdot 0 \rangle_i^{m, \vec{m}}
$$

(similarly for 1x),

$$
= ((h (0x, \vec{y}) \cdot 0) \triangle h (0x, \vec{y}))_1^{m, \vec{m}} = \ell \text{CRN}[h] (x, \vec{y})_1^{m, \vec{m}}.
$$
and for \(1 < i \leq m + 1\),

\[
\langle \ell \text{CRN}[h](0x, \vec{y}) \rangle_i^{m,\vec{n}} = \langle h(z, \vec{y}) \cdot 0 \rangle_1^{m+2-i,\vec{n}} \left[ \langle 0x \rangle_{j+i-1}^{m} / \langle z \rangle_{j}^{m+2-i} \right]_{1 \leq j \leq m+2-i} \\
= \langle h(z, \vec{y}) \cdot 0 \rangle_1^{m+1-(i-1),\vec{n}} \left[ \langle 0x \rangle_{j+(i-1)-1}^{m} / \langle z \rangle_{j}^{m+1-(i-1)} \right]_{1 \leq j \leq m+1-(i-1)} \\
= \langle \ell \text{CRN}[h](x, y) \rangle_{i-1}^{m,\vec{n}} \\
= \langle (h(0x, \vec{y}) \cdot 0) < h(0x, \vec{y}) \rangle \cdot \ell \text{CRN}[h](x, \vec{y}) \rangle_i^{m,\vec{n}}
\]

(similarly for \(1x\)).

(b) Similarly to 11a.

12. For all \(1 \leq i \leq \text{len} \text{TRN}[g, h, h_{\ell}, h_r](x, z, \vec{y})(m, p, \vec{n})\), and by the remark above,

\[
\langle x \triangleleft 1 ? (g(x, z, \vec{y}), t, t) \rangle_i^{m,\vec{n}} = \begin{cases} 
\langle g(x, z, \vec{y}) \rangle_i^{p,\vec{n}} & \text{if } m - 1 = 0 \\
\neg \langle x \triangleleft 1 \rangle_i^{m-1} \land \langle 0(t < t) \cdot t \rangle_i^{m,\vec{n}} & \text{if } m - 1 > 0 \\
\lor (\langle x \triangleleft 1 \rangle_i^{m-1} \land \langle 0(t < t) \cdot t \rangle_i^{m,\vec{n}}) & \text{if } m - 1 > 0
\end{cases}
\]

\[
\leftrightarrow \begin{cases} 
\langle g(x, z, \vec{y}) \rangle_i^{p,\vec{n}} & \text{if } m - 1 = 0 \\
(t_i^{m,\vec{n}}) & \text{if } m - 1 > 0
\end{cases}
\]

where \(t = h(x, z, \vec{y}), \text{TRN}[g, h, h_{\ell}, h_r](x, h_{\ell}(z), \vec{y}), \text{TRN}[g, h, h_{\ell}, h_r](\triangleright x, h_r(z), \vec{y})\).

4.4.2 Rules of inference

For all the rules in Definition 3.1.3, if one of the premises contains an equation of the form \([t = u]^{\vec{m}}\) that degenerates to \(\perp\) because \(\text{len}_{\ell}(\vec{m}_0) \neq \text{len}_u(\vec{m}_1)\), then the rule becomes trivial. We therefore assume that none of the propositional translations of atomic formulas of \(T_1\) are degenerate cases. Also, when we use the notation "\(\bigwedge\)" with no subscript, we implicitly assume that the conjunction is over all relevant values of the index of the term formulas involved.

0. Any standard, complete set of rules for the propositional calculus can be \(p\)-simulated within any \(\mathcal{F}\)-system.

1. We have short \(\mathcal{F}\)-proofs of \([A]^{m,\vec{n}}\). Substituting \((t_i^{\vec{p}})\) for \(\langle x \rangle_i^{m}\) throughout these proofs yield short \(\mathcal{F}\)-proofs of \([A[t/x]]^{\vec{p},\vec{n}}\).
2. (a) First, a few observations. Let \( F = (x = \varepsilon \lor x = 0 \cdot \triangleright x \lor x = 1 \cdot \triangleright x) \). Then,

\[
[x = \varepsilon]^m = \begin{cases} 
\top & \text{if } m = 0 \\
\bot & \text{if } m > 0
\end{cases}
\]

\[
[x = 0 \cdot \triangleright x]^m = \bigwedge_{1 < i} \langle x \rangle_i^m \leftrightarrow \langle 0 \cdot \triangleright x \rangle_i^m
\]

\[
= \langle x \rangle_1^m \leftrightarrow \langle 0 \rangle_1 \land \bigwedge_{1 < i} \langle x \rangle_i^m \leftrightarrow \langle \triangleright x \rangle_i^{m-1}
\]

\[
= \langle x \rangle_1^m \leftrightarrow \bot \land \bigwedge_{1 < i} \langle x \rangle_i^m \leftrightarrow \langle x \rangle_i^m
\]

\[
[x = 1 \cdot \triangleright x]^m = \langle x \rangle_1^m \leftrightarrow \top \land \bigwedge_{1 < i} \langle x \rangle_i^m \leftrightarrow \langle x \rangle_i^m
\]

so that

\[
[F]^m = \begin{cases} 
\top \lor \bot \lor \bot & \text{if } m = 0 \\
\bot \lor \langle \langle x \rangle_1^m \leftrightarrow \bot \land \bigwedge_{1 < i} \langle x \rangle_i^m \leftrightarrow \langle x \rangle_i^m \rangle \rangle \lor \langle \langle x \rangle_1^m \leftrightarrow \top \land \bigwedge_{1 < i} \langle x \rangle_i^m \leftrightarrow \langle x \rangle_i^m \rangle \rangle & \text{if } m > 0
\end{cases}
\]

\[
\leftrightarrow \begin{cases} 
\top & \text{if } m = 0 \\
\langle \langle x \rangle_1^m \leftrightarrow \bot \lor \langle x \rangle_1^m \leftrightarrow \top \rangle \rangle \land \bigwedge_{1 < i} \langle x \rangle_i^m \leftrightarrow \langle x \rangle_i^m & \text{if } m > 0
\end{cases}
\]

\[
\leftrightarrow \begin{cases} 
\top & \text{if } m = 0 \\
\neg \langle x \rangle_1^m \lor \langle x \rangle_1^m & \text{if } m > 0
\end{cases}
\]

Therefore, \([F]^m\) has linear-size \(F\)-proofs.

Now, we have short \(F\)-proofs of

\[
A_\varepsilon = [A[\varepsilon]]^{\vec{n}},
\]

\[
A_0 = [A[x]]^{m,\vec{n}} \rightarrow [A[0x]]^{m,\vec{n}},
\]

and \(A_1 = [A[x]]^{m,\vec{n}} \rightarrow [A[1x]]^{m,\vec{n}}\).

If \(m = 0\), then by Axiom [14] there are short proofs of

\[
[x = \varepsilon]^m \rightarrow ([A]^{m,\vec{n}} \leftrightarrow [A[\varepsilon]]^{\vec{n}}),
\]

which shows that there are short proofs of

\[
[x = \varepsilon]^m \rightarrow ([A[\varepsilon]]^{\vec{n}} \rightarrow [A]^{m,\vec{n}}).
\]

If \(m > 0\), then substituting \(1 \triangleright x\) for \(x\) in \(A_0\) gives short proofs of

\[
[A[1 \triangleright x]]^{m,\vec{n}} \rightarrow [A[0 \cdot \triangleright x]]^{m,\vec{n}};
\]
moreover, by Axiom 1d, there are short proofs of
\[ [x = 0 \cdot x]^m \rightarrow ([A]^m, \vec{n} \leftrightarrow [A[0 \cdot x]]^m, \vec{n})], \]
and therefore, by transitivity, of
\[ [x = 0 \cdot x]^m \rightarrow ([A[1 \cdot x]]^m, \vec{n} \rightarrow [A]^m, \vec{n}). \]
A similar argument shows that there are short proofs of
\[ [x = 1 \cdot x]^m \rightarrow ([A[1 \cdot x]]^m, \vec{n} \rightarrow [A]^m, \vec{n}), \]
which, together with (4.4.1), implies that there are short proofs of
\[ [F]^m \rightarrow ([A[1 \cdot x]]^m, \vec{n} \rightarrow [A]^m, \vec{n}). \]
Applying modus ponens to this and \([F]^m\) gives short proofs of
\[ [A[1 \cdot x]]^m, \vec{n} \rightarrow [A]^m, \vec{n}. \quad (4.4.2) \]
Repeated substitutions of \(1 \cdot x\) for \(x\) in the proof of (4.4.2) give short proofs of
\[ [A[11 \cdot x]]^m, \vec{n} \rightarrow [A[1 \cdot x]]^m, \vec{n}. \]
\[ \vdots \]
\[ [A[\hat{m} \cdot x]]^m, \vec{n} \rightarrow [A[\hat{m} - 1 \cdot x]]^m, \vec{n} \]
(where we remind the reader that \(\hat{k}\) is a shorthand for \(1 \cdots 1\)).
Since \(\hat{m} \cdot x = \varepsilon\), using Axiom 1d and modus ponens gives short proofs of
\[ [A[\varepsilon]]^\vec{n} \rightarrow [A[\hat{m} \cdot x]]^m, \vec{n}, \]
and using transitivity \(m + 1\) times now gives short proofs of
\[ [A[\varepsilon]]^\vec{n} \rightarrow [A]^m, \vec{n}. \]
A final application of modus ponens with \(A_\varepsilon\) gives the short proofs of \([A]^m, \vec{n}\) we wanted: as can easily be seen, the size of this proof is \(O(m \cdot p(m, \vec{n}))\), where \(p(m, \vec{n})\) was the maximum size of the proofs of \(A_\varepsilon, A_0, A_1\).
(b) The same reasoning as for part (a) applies.
3. We have short proofs of $[[A[x, z]]=p, \vec{n}]]$, $[[A[0, z]]=p, \vec{n}]]$, $[[A[1, z]]=p, \vec{n}]]$, and

$$A' = ([[A[x, h(z)]]]^{m, p, \vec{n}} \land [[A[x', h(z)'(z)]]]^{m, p, \vec{n}}) \rightarrow [[A[x, z]]=^{m, p, \vec{n}}].$$

If $m = 0$, using modus ponens twice on formula (4.4.1) gives a short proof of $[[A]]=^{m, p, \vec{n}}$. If $m > 0$, then by repeatedly substituting first $x, h(z)$ and then $x', h(z)$ for $x, z$ in the proof of $A'$, we get a binary tree of short proofs of formulas of the form of $A'$, where the formula at the root is $[[A[x, z]]=^{m, p, \vec{n}}$, the formula at each node is implied by the conjunction of the formulas at its children nodes, and at the leaves, the terms being substituted for $u$ can all be proved to be equal to 0 or 1 (single bits of $x$).

For example, if $m = 3$, the tree would have the form depicted in Figure 4.4.1 (where we’ve indicated only the consequent of the formula being proved at each node, so that a node $B$ with children $C$ and $D$ represents a proof of the formula $(C \land D) \rightarrow B$ and a node $B$ with one child $C$ represents a proof of the formula $C \rightarrow B$).

\begin{center}
\begin{tikzpicture}
  \node (root) at (0,0) {A[x, z]};
  \node (left) at (-2, -2) {A[x, h(z)]};
  \node (right) at (2, -2) {A[x, h(z)]};
  \node (left-left) at (-4, -4) {A[0, z'] \land A[1, z']};
  \node (left-right) at (-4, -4) {A[(x), h(h(z))]};
  \node (right-left) at (4, -4) {A[0, z'] \land A[1, z']};
  \node (right-right) at (4, -4) {A[0, z'] \land A[1, z']};
  \draw (root) -- (left);
  \draw (root) -- (right);
  \draw (left) -- (left-left);
  \draw (left) -- (left-right);
  \draw (right) -- (right-left);
  \draw (right) -- (right-right);
\end{tikzpicture}
\end{center}

Figure 4.4.1: Proof tree for $m = 3$.

Therefore, the proofs of $[[A[0, z]]=p, \vec{n}]]$ and $[[A[1, z]]=p, \vec{n}]]$ can be used with Rule III (substituting the right terms for $z$) and modus ponens to prove the formulas at the first level, and going up level by level using modus ponens, we obtain a proof of the consequent of the formula at the root of the tree, i.e., $[[A]]=^{m, p, \vec{n}}$. Moreover, if $q(m, p, \vec{n})$ is the maximum proof size of the premises and $c$ is a constant satisfying $|h(z)| \leq 2^c |z|$ and $|h(z)'| \leq 2^c |z|$, then the size of this proof is $O(m \cdot q(m, p \cdot m, c, n))$ since the tree has depth no more than $\log m$ and thus size no more than $2m$.

**Remark 4.4.1** Note that the estimates on the size of $F$-proofs for $T_1$’s theorems given above might be used to get more precise upper bounds than are currently known for the size of $F$-proofs of certain families of tautologies. For example, the family of tautologies arising from the pigeonhole principle was first shown to have poly-size $F$-proofs by Buss [8] (whose estimate of the size of the proofs was $O(n^{20})$); a careful analysis of the $T_1$-proof given in Chapter together
with the results of this chapter could provide a better bound (the details would be somewhat tedious but straightforward), and the same could be done for the other families of tautologies mentioned at the end of Chapter 3.
Chapter 5

$T_1$ Proves the Soundness of $\mathcal{F}$

In this chapter, we will show how to formalize a particular $\mathcal{F}$-system in $T_1$, how to formalize Buss’s algorithm for the Boolean Sentence Value Problem (BSVP) \cite{10} in $T_1$, and how to use the BSVP algorithm to prove the soundness of the given $\mathcal{F}$-system in $T_1$. Then, we show that $\mathcal{F}$ provably $p$-simulates any proof system $S$ whose soundness can be proved in $T_1$.

5.1 Formalizing $\mathcal{F}$-systems

Because any two $\mathcal{F}$-systems $p$-simulate each other, we will focus on the particular $\mathcal{F}$-system below:

**language:** variables $p_1, p_2, \ldots$, constants $\top$ and $\bot$, connective $\rightarrow$, brackets $( )$

**formulas:** $\top$, $\bot$, $p_i$ (for any $i \geq 1$), and recursively, $(A \rightarrow B)$ for any formulas $A$ and $B$

**axiom schemes** (for any formulas $A$, $B$, $C$):

1. $(A \rightarrow (B \rightarrow A))$
2. $((A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C)))$
3. $(((B \rightarrow \bot) \rightarrow (A \rightarrow \bot)) \rightarrow (A \rightarrow B))$
4. $\top$

**rule** (modus ponens): $A, (A \rightarrow B) \vdash B$

5.1.1 Formulas

Given a formula $A$ of $\mathcal{F}$, we will encode $A$ into a string $\#_{\ell}A$ in the following way: Let $w_A = \max\{i : p_i \text{ appears in } A\}$ and $\ell_A = 1 + \lfloor \log(w_A + 1) \rfloor$ (one more than the binary length of $w_A$). Then, for $2^\ell \geq \ell_A + 2$, $\#_{2^\ell}A$ is obtained from $A$ by using the following Gödel-numbering scheme
where \((i)_{\frac{2^\ell}{2}} \in \{0,1\}_{2^\ell/2}\) represents \(i\) in binary using exactly \(2^\ell - 2\) bits—note that because of our choice of \(2^\ell\), this will always start with a 0).

| symbol | code |
|--------|------|
| \(p_i\) | 00\((i)_{\frac{2^\ell}{2}}\) |
| \(\bot\) | 0010\(2^\ell-3\) |
| \(\top\) | 0011\(2^\ell-3\) |
| \(\rightarrow\) | 100\(2^\ell-2\) |
| \()\) | 010\(2^\ell-2\) |

In fact, because we can count in \(T_1\), it is possible to define a slightly more complicated encoding where the codes for “(”, “\(\rightarrow\)”, and “)” include information about the logical depth of the symbol (making sure that \(\ell_A\) is adjusted to be the maximum of its old value and the logical depth of \(A\)). In what follows, we will use “(\(j\)” “\(\rightarrow j\)”, and “\(j\)” to represent the corresponding symbol at a logical depth of \(j\).

For example, the formula \(A = (p_2 \rightarrow ((p_{10} \rightarrow \bot) \rightarrow p_2))\) can be rewritten as \(A = (0p_2 \rightarrow_0 (1(2p_{10} \rightarrow_2 \bot) \rightarrow_1 p_2)_1)_0\) by including the depth information for each symbol except logical variables and constants, which would be encoded as follows (where the string was split in two to fit on the page, and a little bit of space was added between blocks of bits representing each symbol, for readability):

\[
\#_{8} A = 100000000 00000010 11000000 10000001 10000010 00001010 \cdots \\
\cdots 11000010 00100000 01000010 11000001 00000010 01000001 01000000.
\]

With this encoding, it is easy to write a function \(\text{formula}(x, z)\) in \(T_1\) that returns 1 if \(x = \#_{2^\ell} A\) for some formula \(A\) such that \(|\text{pow}^L(z)| = 2^\ell \geq \ell_A + 2\), or 0 otherwise. The function is defined by \(r\text{pow}CRN\) and simply checks that \(x\) is one of “\(\#_{2^\ell} \bot\)”, “\(\#_{2^\ell} \top\)”, “\(\#_{2^\ell} p_i\)”, or that \(x\) has the form “\((a \cdots)_a\)” and that each symbol in \(x\) is preceded by a valid string of symbols, according to the following simple rules (using counting and masking operations):

- “\(p_i\)”, “\(\bot\)”, and “\(\top\)” must immediately follow either “(” or “\(\rightarrow\)”;
- “(\(j\)” must immediately follow either “(\(j-1\)” or “\(\rightarrow j-1\)”;
- “\(\rightarrow j\)” must either immediately follow “(\(j\)\(p_i\)” “(\(j\)\(\bot\)” or “(\(j\)\(\top\)”, or it must follow, in order, “(\(j\)(\(j+1\)\(\cdots\))\(j+1\)” (where everything between the parentheses has depth at least \(j + 1\));
- “\(j\)” must either immediately follow “\(\rightarrow j\)\(p_i\)”, “\(\rightarrow j\)\(\bot\)” or “\(\rightarrow j\)\(\top\)”, or it must follow, in order, “\(\rightarrow j\)(\(j+1\)\(\cdots\))\(j+1\)” (where everything between the parentheses has depth at least \(j + 1\)).
5.1.2 Proofs

Now, we can encode $\mathcal{F}$-proofs easily. A proof $A_1, \ldots, A_k$ is encoded by a pair of strings: $(\#_{2^\ell} A_1, \ldots, \#_{2^\ell} A_k)_k$ and $(j_1, \ldots, j_k)_k$, where $2^\ell \geq \max_{1 \leq i \leq k} \{ \ell_{A_i} \} + 2$ and

$$j_i = \begin{cases} 
(0, m) & \text{if } A_i \text{ is an instance of axiom } m, \\
(k_1, k_2) & \text{if } A_{k_2} = A_{k_1} \rightarrow A_i.
\end{cases}$$

Again, it is straightforward to write a function $\text{proof}(x, y, z, w)$ in $T_1$ that returns 1 if $x, y$ encode an $\mathcal{F}$-proof for $2^\ell = |\text{pow}^L(z)|$ and $k = |w|$, or 0 otherwise, by using simple masking operations and rpowCRN. (Technically speaking, the encodings of the formulas in a proof must be padded so they all have the same length, but it is a simple matter to take care of.)

Finally, we can define in $T_1$ the following function:

$$F(x, y, z, w) = \begin{cases} 
|w| \pi_{|w|}(x) & \text{if } \text{proof}(x, y, z, w) = 1, \\
\#_{|\text{pow}^L(z)|} \top & \text{otherwise},
\end{cases}$$

that returns the tautology proved by $x, y$ (or some fixed tautology if $x, y$ is not an $\mathcal{F}$-proof).

5.2 Buss’s algorithm for the BSVP

For reference purposes, we will now summarize Buss’s most recent published $A\text{LOGTIME}$ algorithm for the BSVP [10]. Actually, we present a slight variation of his algorithm applied to formulas containing only the connective “$\rightarrow$” as opposed to “$\land$” and “$\lor$”. Given such a Boolean sentence, we can represent it as a binary tree with $2^{d+1} - 1$ leaves for some $d$ (we can pad any sentence so that it meets this condition by preceding it with enough copies of “$\top \rightarrow \cdots$”), where each leaf stores either “$\top$” or “$\bot$” and each interior node represents the connective “$\rightarrow$”. For two nodes $U$ and $V$ in this tree, we write “$U \triangleright V$” to mean that $U$ is an ancestor of $V$, and “$U \triangleright V$” to mean $U = V$ or $U \triangleright V$. The least common ancestor of $U$ and $V$ is denoted $\text{lca}(U, V)$. By convention, we draw trees with the root at the top and the leaves at the bottom, so that “above” and “below” correspond to “ancestor” and “descendant”, respectively. Also, we define a scarred sentence as a binary tree whose leaves store $\top$ or $\bot$ and that contains exactly one internal node with only one child (the missing child is called the scar). The “value” of a scarred sentence is defined to be a pair of truth-values $(t_\top, t_\bot)$, where $t_\top$ is the value of the Boolean sentence obtained when the scar is replaced by $\top$, and similarly for $t_\bot$.

The algorithm will be described as a pebbling game on the formula’s tree between two players: the Pebbler and the Challenger, and it proceeds in rounds. During each round, the Pebbler places pebbles labelled with a truth-value “0” or “1” on nodes of the tree, representing
assertions by the Pebbler that the subformulas rooted at those nodes have the indicated truth-values. Following the Pebbler’s move, the Challenger \textit{challenges} one of the pebbled positions $U$, representing an assertion by the Challenger that the pebble value at $U$ is incorrect (and implicitly, that every pebbled position below $U$ is correct).

Intuitively, the essential feature of the pebbling-challenging game is to break up the work by creating scarred subsentences and evaluating them at the same time as their scar, instead of performing the evaluation sequentially. (For example, we could evaluate $(A \to B) \to C$ by evaluating $(A \to \star) \to C$ in parallel with $B$, where “$\star$” indicates the position of the scar.) Together with Buss’s innovative technique for finding scar positions in a semi-oblivious fashion through distinguished leaves, this feature of the algorithm allows even unbalanced sentences to be evaluated in a logarithmic number of steps.

The game is designed so that the Pebbler has a winning strategy if the value of the sentence is “1”; otherwise, the Challenger has a winning strategy. Many of the rules of the game might seem somewhat arbitrary and strict, but they are designed so that a play of the game can be evaluated in $\text{ALOGTIME}$ while preserving the property that the correct player has a winning strategy. For example, the game will never last more than $d$ rounds (when there are $2^{d+1} - 1$ leaves), and since specifying arbitrary pebble positions would require $O(d)$ bits (which would take us outside $\text{ALOGTIME}$), there must be a “semi-oblivious” way of specifying pebble positions using only a constant number of bits per round.

Before giving the details of the pebbling game, we need to introduce a bit more notation. First, leaves will be numbered from left to right, starting with 1, and assigned a \textit{rank} equal to the largest integer $k$ such that $2^k$ divides the leaf number. Next, in each round $i \geq 1$, there will be distinguished leaves $L_i$, $C_i$, and $R_i$ (for “left”, “center”, and “right”, respectively) and distinguished nodes $A_i$ and $B_i$ (for “above” and “below”, respectively), satisfying the following conditions (see Figure 5.2.1 for a picture).

1. $A_i \geq B_i \geq C_i$, with $A_i = B_i$ only if $B_i = C_i$;

2. $A_i$ is the lowest (and latest) challenged node, while $B_i$ is the highest pebbled position satisfying the first condition—or $B_i = C_i$ if there is no pebbled node below $A_i$ (informally, the players have “agreed” at $B_i$ but “disagree” at $A_i$);

3. $L_i$ and $R_i$ are distinct leaves of rank $d - i$ and $C_i$ is of rank greater than $d - i$;

4. every leaf in the subtree rooted at $A_i$ but outside the subtree rooted at $B_i$ has number in the range

   $$(L_i - 2^{d-i}, L_i + 2^{d-i}) \cup (R_i - 2^{d-i}, R_i + 2^{d-i})$$

(where the intervals are open).
5.2. Buss’s algorithm for the BSVP

The pebbling game proceeds as follows. In round 0, the Pebbler must pebble the root node with value “1”, and the Challenger must challenge the pebble at the root. In preparation for round 1, set $A_1$ to be the root, $B_1 = C_1$ to be the leaf numbered $2^d$, and $L_1$ and $R_1$ to be the leaves numbered $2^{d-1}$ and $2^d + 2^{d-1}$, respectively (see Figure 5.2.2 for an example where each leaf’s number and rank is indicated). In round $i \geq 1$, let $U_i = \text{lca}(L_i, C_i)$ and $V_i = \text{lca}(C_i, R_i)$ (note that $U_i$ and $V_i$ are distinct because $L_i$ and $R_i$ are distinct).

Figure 5.2.2: Labelled example of the BSVP algorithm at round 1.
In each round, the Pebbler uses six bits of information (one per node) to pebble $U_i$, $V_i$, and their two immediate children ($U_i^1$, $V_i^1$ to the left and $U_i^2$, $V_i^2$ to the right, respectively). In addition, the Pebbler must use three bits of information to specify the relative positions of $A_i$, $U_i$, and $V_i$ (i.e., which nodes are ancestors of which ones—since all three nodes are ancestors of $C_i$, they all lie on the path from $C_i$ to the root).

The Challenger then challenges one node from among $A_i$, $U_i$, $U_i^1$, $U_i^2$, $V_i$, $V_i^1$, or $V_i^2$ (using three bits to specify which one), subject to the conditions that the node challenged must be in the subtree rooted at $A_i$ and outside the subtree rooted at $B_i$.

For round $i + 1$, $A_{i+1}$ is set to the node just challenged, $B_{i+1}$ is set to the highest pebbled node below $A_{i+1}$ (or to $C_{i+1}$ if there is no pebbled node below $A_{i+1}$), and $L_{i+1}, C_{i+1}, R_{i+1}$ are set according to Table 5.2.1.

| Challenged Node | Pebbler says $U_i \triangleright V_i$ | Pebbler says $V_i \triangleright U_i$ |
|-----------------|-------------------------------------|-------------------------------------|
| $U_i^1$        | $C_{i+1} = L_i$                    | $C_{i+1} = L_i$                    |
|                | $R_{i+1} = L_i + 2d^{i-1}$         | $R_{i+1} = L_i + 2d^{i-1}$         |
|                | $L_{i+1} = L_i - 2d^{i-1}$         | $L_{i+1} = L_i - 2d^{i-1}$         |
| $U_i^2$        | $C_{i+1} = C_i$                    | $C_{i+1} = C_i$                    |
|                | $R_{i+1} = R_i + 2d^{i-1}$         | $R_{i+1} = R_i - 2d^{i-1}$         |
|                | $L_{i+1} = L_i + 2d^{i-1}$         | $L_{i+1} = L_i + 2d^{i-1}$         |
| $V_i^1$        | $C_{i+1} = C_i$                    | $C_{i+1} = C_i$                    |
|                | $R_{i+1} = R_i - 2d^{i-1}$         | $R_{i+1} = R_i - 2d^{i-1}$         |
|                | $L_{i+1} = L_i + 2d^{i-1}$         | $L_{i+1} = L_i - 2d^{i-1}$         |
| $V_i^2$        | $C_{i+1} = R_i$                    | $C_{i+1} = R_i$                    |
|                | $R_{i+1} = R_i + 2d^{i-1}$         | $R_{i+1} = R_i + 2d^{i-1}$         |
|                | $L_{i+1} = R_i - 2d^{i-1}$         | $L_{i+1} = R_i - 2d^{i-1}$         |
| $U_i$ or $V_i$ | Game Ends                          | Game Ends                          |

Table 5.2.1: Next leaf nodes in Buss’s BSVP algorithm.
5.2. Buss’s algorithm for the BSVP

Now, it is easy to show by induction on the number of rounds played that properties 1–4 are preserved for the duration of the algorithm: it is simply a matter of checking case-by-case each possibility in Table 5.2.1 for the values of $C_i$, $L_i$, and $R_i$. For example, suppose that $V_i^2$ is challenged in Figure 5.2.2, then for round 2, $A_2 = V_2^2$, $B_2 = C_2 = R_1 = 24$, $L_2 = R_1 - 4 = 20$, and $R_2 = R_1 + 4 = 28$ (we refer to leaves by their number), so $A_2 \triangleright B_2 = C_2$, $A_2$ is the lowest challenged node and $B_2 = C_2$, $L_2$ and $R_2$ have rank $2 = 4 - 2$ and $C_2$ has rank $3 > 2$, and every leaf in the subtree rooted at $A_2$ but outside the subtree rooted at $B_2$ has number in the range $(20 - 2^4 + 2, 20 + 2^4 - 2) \cup (28 - 2^4 + 2, 28 + 2^4 - 2) = (16, 24) \cup (24, 32)$.

The game ends as soon as one of the players makes an “obvious” mistake, *i.e.*, one from the following list. (Note that by property 4, the game must end by round number $d$ because $(L_d - 2^d, L_d + 2^d - 2) \cup (R_d - 2^d, R_d + 2^d - 2) = \{L_d, R_d\}$; it is easy to see that in that case, one of the two players will be forced to make a mistake from the list below.)

- **Pebbler**: when the input nodes of a gate are either leaves or are pebbled and the output node is pebbled incompatibly.
- **Challenger**: when the output of a gate whose input nodes are either leaves or pebbled is challenged, even though it is correctly pebbled.
- **Pebbler**: when a leaf is incorrectly pebbled.
- **Challenger**: when a correctly pebbled leaf is challenged.
- **Pebbler**: when a node is pebbled with both “0” and “1”.
- **Pebbler**: when an incorrect assertion is made about whether $U_i \triangleright V_i$, $A_i \triangleright U_i$, $A_i \triangleright V_i$.
- **Challenger**: when the challenged node is above a previously challenged node.
- **Challenger**: when the challenged node is at or below a previously agreed upon pebble value (a pebble is “agreed upon” if it was placed in an earlier round and in that round, the Challenger challenged an ancestor of that pebble).

It is straightforward to see that the game produces the correct result: if the value of the sentence is “true”, the Pebbler can win the game by simply pebbling every node with its correct value and making assertions compatible with the structure of the sentence, while if the value of the sentence is “false”, the Challenger can win the game by always challenging the lowest incorrectly pebbled node that is not below a previously agreed upon node.

Moreover, the game can be translated into an $A\log t i m e$ algorithm, as follows: First, simulate possible plays of the game using existential moves for the Pebbler and universal moves for the Challenger. Then, for each such game, existentially guess the first mistake made and
universally verify that no earlier mistake was made. Note that from a play of the game, it is easy to determine the last round when \( L_j \) was computed from \( R_j \) and to compute the appropriate sum of powers of 2 to add to \( R_j \) in order to get \( L_i \) (the same goes for \( R_i \)). As for \( C_i \), simply find the last round when \( C_j \) was equal to \( L_j \) or \( R_j \) and we know that \( C_i = C_j \). Finally, because it is possible to count in \( ALOGTIME \), ancestors can readily be computed to find \( U_i \), \( V_i \) and thus determine \( A_i \) and \( B_i \).

5.3 Formalizing the BSVP

The algorithm that we will use to solve the BSVP in \( T_1 \) is simply a formalization inside our theory of the algorithm described in the previous section. To formalize this algorithm inside \( T_1 \), we will define a function \( BSVP \) that decomposes and evaluates the sentence, using TRN to perform the work in parallel, disjunction (OR) over all possible Pebbler guesses to emulate existential moves, and composition and implication to emulate Challenger’s universal moves.

First, it is easy to define by \( rpowCRN \) a function \( sentence(v, x, z) \) that takes as argument a truth-value assignment \( v \) (represented simply by a string whose 1st bit is the value of \( p_1 \), whose 2nd bit is the value of \( p_2 \), etc.) and the encoding of a formula \( x = \#[powL(z)], A \), and returns the sentence obtained by substituting the given truth-values for the variables in \( x \).

Next, we define the function \( BSVP(d, h, m, s) \) that does the work according to Buss’s algorithm. There will be one variation: because we want the function to apply to arbitrary sentences, but a sentence must have a power of 2 minus 1 leaves in the algorithm, we will pad sentences so they have \( 2^{d+1} - 1 \) leaves and remember the position of the root of the original sentence inside the padded version as “\( M \)”. (The algorithm needs to be changed so that it takes the distinguished node \( M \) into account at the same time as \( A, B, L, C, R \), but the changes are easy to make since \( M \) remains fixed for the duration of the algorithm.) In what follows, the parameter \( s \) is fixed and encodes a superformula of the Boolean sentence we are evaluating (padded so it always has a power of two minus one leaves), the parameter \( m \) is fixed and indicates the root \( M \) of the subformula of \( s \) whose value we are interested in, the parameter \( h \) varies and represents the history of the game so far, as a sequence of blocks \( b_1 a_1 \cdots b_i a_i \) (each block a constant-length string encoding Pebbler’s guesses on the relative positions of the nodes \( A, U, V, B \), and \( M \) (in \( b_j \)), as well as Challenger’s chosen node (in \( a_j \))), and \( d \) varies and represents 2 to the power of the current round number, in unary. The function \( BSVP(d, h, m, s) \) returns a truth-triplet \( (c, t_{\top}, t_{\bot}) \), where \( c \) is a check-bit indicating whether \( h \) is a valid description of the structure of \( s \) or not, and \( (t_{\top}, t_{\bot}) \) is the value of the possibly scarred subformula of \( M \) picked out by \( h \). The function will be defined by TRN on \( d \), following Buss’s algorithm, but first, we must specify how truth-triplets can be combined in various ways.
We generalize disjunction and implication to truth-triplets, and define composition of truth-triplets, as follows:

\[
(c^1, t^1_t, t^1_{\perp}) \land (c^2, t^2_t, t^2_{\perp}) = (c^1 \land c^2, (c^1 \land t^1_t) \land (c^2 \land t^2_t), (c^1 \land t^1_{\perp}) \land (c^2 \land t^2_{\perp})),
\]

\[
(c^1, t^1_t, t^1_{\perp}) \rightarrow (c^2, t^2_t, t^2_{\perp}) = (c^1 \land c^2, t^1_t \rightarrow t^2_t, t^1_{\perp} \rightarrow t^2_{\perp}),
\]

\[
(c^1, t^1_t, t^1_{\perp}) \circ (c^2, t^2_t, t^2_{\perp}) = (c^1 \land c^2, t^1_t \circ t^2_t, t^1_{\perp} \circ t^2_{\perp})
\]

(where \(t^1_t\) is equal to \(t^2_{\perp}\) if \(t^2_t = \top\) and \(t^1_{\perp}\) if \(t^1_t = \bot\), and similarly for \(t^2_t\)).

Then, for \(h = b_1 a_1 \cdots b_n a_i\), we can define BSVP\((d, h, m, s)\) as follows:

\[
\text{BSVP}(d, h, m, s) = \bigvee_{\text{all Pebbler guesses } b} \left[ \begin{array}{c}
\text{composition of BSVP}(\uparrow d, hba, m, s)’s using } \circ \text{ and } \rightarrow, \\
\text{based on structure induced by } b \text{ and where } a \text{ picks out }
\end{array} \right].
\]

Because \(b\) has a fixed length, the disjunction actually represents a fixed number of cases, each one of which has a unique structure determined by the value of \(b\). We will not list all possible cases here (they can easily be written down from the description of Buss’s algorithm and Table 5.2.1), but we give two illustrative examples based on the sentence depicted in Figure 5.3.2.

1. Consider the sentence depicted in Figure 5.3.1, where we have filled-in the unique interior node that represents “\(m\)” and each leaf that falls inside the correct intervals around \(L_1\) and \(R_1\). At round 1, we have that BSVP\((d, h, m, s) = \text{BSVP}(\uparrow d, hba_{U1}, m, s) \rightarrow \text{BSVP}(\uparrow d, hba_{U2}, m, s)\), where “\(a_{U1}\)” and “\(a_{U2}\)” are fixed-length strings representing which subsentence is selected, and “\(b\)” is the unique fixed-length string representing the correct structure of the formula. The other parts of the formula (under \(V^2_t\)) are of no interest because they fall outside “\(m\)”.

![Figure 5.3.1: Labelled example of the BSVP algorithm at round 1.](image)

2. If we look at the first recursive call of BSVP in the preceding case, we have the situation depicted in Figure 5.3.2, in which case it is easy to see that BSVP\((d, h, m, s) = \)
Chapter 5. \( T_1 \) Proves the Soundness of \( \mathcal{F} \)

\[
\text{BSVP}(d, hba_A, m, s) \circ (\text{BSVP}(d, hba_{U_1}, m, s) \to (\text{BSVP}(d, hba_{U_2}, m, s) \circ (\text{BSVP}(d, hba_{V_1}, m, s) \to \text{BSVP}(d, hba_{V_2}, m, s))))).
\]

At the last round (when \( d = 1 \)), the history \( h \) is analyzed and compared with the actual structure of the sentence, and the check-bit \( c \) returned is \( \top \) iff they agree. The check-bit can be obtained by taking the OR of a bit-string computed by \( \text{rpowCRN} \) on \( h \), where the bit output for each block \( b \) in \( h \) is 1 iff \( b \) is correct. Moreover, each of these bits can be computed by finding the positions of the \( L \), \( C \), and \( R \) leaves from the first part of the history \( h \) (which is easy to do by TRN) and then finding least common ancestors of these leaves (which again can be done easily by TRN).

At the same time, the actual sentence left at the last round will have one of the simple forms shown in Figure 5.3.3, which can all be evaluated trivially since we know the values of \( L \), \( C \), and \( R \), and we can easily check when \( B = C \).

Figure 5.3.3: Base cases for \( \text{BSVP} \).

There remains one technical detail that needs to be taken care of in the definition of \( \text{BSVP} \): as given, the definition is not a proper application of TRN since the value of \( \text{BSVP}(d, h, m, s) \) is recursively defined in terms of \( \text{BSVP}(d, h', m, s) \) for more than one value of \( h' \). The situation can easily be remedied, in the following way. First, the bounded \( \bigvee \) over all values of \( b \) can be implemented with a subtree whose depth is equal to the number of bits in \( b \) (which is a constant), where a single bit is added to \( b \) at every level (to get all possible values of \( b \) at the bottom) and
5.3. Formalizing the BSVP

\( \mathcal{W} \) is used to combine the values at each node in the subtree. Next, the composition of values based on the structure determined by \( b \) can similarly be carried out step-by-step using a subtree of constant depth, where at each step, the value of \( b \) is used to determine which function (\( \circ \) or \( \rightarrow \)) to apply to combine the results, and which bits to add to \( a \) to select subformulas. Both of these steps only require increasing the length of \( d \) by a constant factor, and using \( \text{powmod}^k \) to determine which level is being evaluated in the subtrees.

Finally, we can define a function \( \text{VALUE}(v, x, z) \) that computes the truth-value of the formula encoded by \((x, z)\) under \( v \):

\[
\text{VALUE}(v, x, z) = \text{AND}(\text{BSVP}(2^k \times \text{leaves}(x, z), \varepsilon, \text{mask}(x, z), \text{sentence}(v, \text{pad}(x, z), z))),
\]

where \( \text{pad}(x, z) \) pads the formula encoded by \( x \) and \( z \), adding enough copies of “\( \top \rightarrow \cdots \)” to the left so that it has \( 2^{\delta+1} - 1 \) leaves for some integer \( \delta \geq 0 \), \( \text{mask}(x, z) \), returns the position of the root of the formula encoded by \( x \) inside \( \text{pad}(x, z) \), using a bitmask (where the root is indicated by its main connective “\( \rightarrow \)”), \( \text{leaves}(x, z) \) returns a string of length \( 2^\delta \) (for the same \( \delta \) as above), and \( k \) is the fixed number of bits in one block “\( ba \)” of the history. All these functions are easily defined in \( T_1 \) using \( \text{rpowCRN} \) and \( \text{TRN} \), as follows.

First, note that a formula with \( n \) leaves always contains exactly \( 4n - 3 = n + 3(n - 1) \) symbols (1 for each leaf, and 3 for each connective: two parentheses and one connective), each one encoded by a block of length \( |\text{pow}^k(z)| \). This means that the function

\[
\text{numleaves}(x, z) = \text{powdiv}^k((x \cdot (3 \times \text{pow}^k(z))) \bowtie^2, z)
\]

returns the number of leaves of the formula encoded by \( x \), in unary. Also, we can check whether a formula has a power of 2 minus 1 leaves or not with the function \( \text{ispow}^k \). Hence, if we define

\[
\text{leaves}(x, z) = \text{sevael}(x \cdot (7 \times \text{pow}^k(z)), z),
\]

where

\[
\text{sevael}(x, z) = \begin{cases} 
1 & \text{ if } x \leq^k 8 \times \text{pow}^k(z), \\
\text{sevael}(x \bowtie^2, z) \cdot \text{sevael}(\triangleleft^2 \cdot x, z) & \text{ otherwise},
\end{cases}
\]

then the string \( x \cdot (7 \times \text{pow}^k(z)) \) contains \( 4n - 3 + 7 = 4(n + 1) \) blocks of bits of length \( \text{pow}^k(z) \) so that \( \text{leaves}(x, z) \) contains exactly \( 2^\delta \) bits for any formula \( x \) that contains between \( 2^\delta \) and \( 2^{\delta+1} - 1 \) leaves, inclusive.

Next, we define by \( \text{TRN} \) a function \( \text{padding}(y, w, z) \) that returns a balanced sentence containing exactly \( |y| \) leaves, each one having the value \( \top \), where the logical depth of each symbol is at least \( |w| \) and the length of each symbol’s encoding is \( |\text{pow}^k(z)| \):

\[
\text{padding}(y, w, z) = \begin{cases} 
y \Rightarrow^2 L (\varepsilon, |\text{pow}^k(z)| \top) & \text{ if } y = \varepsilon, 0, 1, \\
|\text{pow}^k(z)|(|w| \bowtie^2 \cdot \text{padding}(y \bowtie^2, w1, z) \cdot |\text{pow}^k(z)| \cdot \text{padding}(\triangleleft^2 y, w1, z) \cdot |\text{pow}^k(z)| |w|) & \text{ otherwise}.
\end{cases}
\]
Now, we can easily define
\[\text{pad}(x, z) = x \quad \text{(if } x \text{ has a power of 2 minus 1 leaves)},\]
\[\text{pad}(x, z) = \#_{\text{pow}(z)}(0 \cdot \text{padding} \text{numleaves}(x, z) \rhd \text{leaves}(x, z), 1, z) \cdot \#_{\text{pow}(z)}(z) \cdot \#_{\text{pow}(z)}(z) \cdot 0 \quad \text{(otherwise),}\]
where \text{deepen}(x, z) is easily defined by rpowCRN to add 1 to the logical depth of every symbol in the formula \(x\). Finally, we can define \text{mask} by rpowCRN:
\[\text{mask}(x, z) = \text{the unique connective } \rightarrow \text{ at logical depth 0 inside pad}(x, z)\]
\[\text{mask}(x, z) = \text{the second connective } \rightarrow \text{ at logical depth 1 inside pad}(x, z)\quad \text{(otherwise)}.

5.3.1 Proof of correctness in \(T_1\)

Theorem 5.3.1 \(T_1\) proves \(\text{VALUE}(v, \#_{\text{pow}(z)}(\top), z) = 1, \text{VALUE}(v, \#_{\text{pow}(z)}(\bot), z) = 0,\) and for arbitrary formulas \(M\) and \(N\),
\[\text{VALUE}(v, \#_{\text{pow}(z)}(M \rightarrow N), z) = \text{VALUE}(v, \#_{\text{pow}(z)}(M, z) \rightarrow^b \text{VALUE}(v, \#_{\text{pow}(z)}(N, z)\]
(i.e., \text{VALUE} is intensional).

Proof. The first two statements follow directly from the definition of the functions involved. The third statement follows from Claim \ref{5.3.1} below. □

Claim 5.3.1

1. If \(h\) picks out a supersentence of \((M \rightarrow N)\), possibly scarred at \(B\), in the sentence encoded by \(s\), then \(\text{BSVP}(d, h, \text{lca}(m, n), s) = \text{BSVP}(d, h, m, s) \rightarrow \text{BSVP}(d, h, n, s)\) (where \text{lca}(m, n) is a mask indicating the position of the least common ancestor of the nodes masked by \(m\) and \(n\) in the sentence \(s\)).

2. \(\text{BSVP}(d, \varepsilon, m, s) = \text{BSVP}(d', \varepsilon, m', s')\) for all values of the parameters that represent the same sentence, i.e., given \(x = \#_A\), for any \(s, s'\) that are supersentences of \(x\), where \(m, m'\) and \(d, d'\) are defined appropriately.

Proof

1. By induction on \(d\). When \(d = 1\), only the first four cases of Figure \ref{5.3.3} apply. Suppose we are in case 4; then, \(A = (M \rightarrow N)\) and \(\text{BSVP}(d, h, \text{lca}(m, n), s) = (c, r, \top)\) (where \(r\)
is the value of node \( R \), \( \text{BSVP}(d, h, m, s) = (c, \top, \bot) \) (since \( M = B \) is the scar), and \( \text{BSVP}(d, h, n, s) = (c, r, r) \), so the statement is true. The other three cases are similar.

Now, if \( h \) picks out a supersentence of \((M \to N)\), possibly scarred at \( B \), then consider the following cases.

(a) If \( \text{BSVP}(d, h, \text{lca}(m, n), s) = \text{BSVP}(\triangleright d, hba, \text{lca}(m, n), s) \), then \( hba \) also picks out a supersentence of \((M \to N)\), possibly scarred at \( B \), and the statement is true by the induction hypothesis.

(b) If \( U > V = (M \to N) \), then

\[
\text{BSVP}(d, h, \text{lca}(m, n), s) = \text{BSVP}(\triangleright d, hba_{V_1}, \text{lca}(m, n), s) \to \text{BSVP}(\triangleright d, hba_{V_2}, \text{lca}(m, n), s).
\]

Also,

\[
\text{BSVP}(d, h, m, s) = \text{BSVP}(\triangleright d, hba_{V_1}, m, s) = \text{BSVP}(\triangleright d, hba_{V_1}, \text{lca}(m, n), s)
\]

and

\[
\text{BSVP}(d, h, n, s) = \text{BSVP}(\triangleright d, hba_{V_2}, n, s) = \text{BSVP}(\triangleright d, hba_{V_2}, \text{lca}(m, n), s)
\]

by the induction hypothesis. Hence, the statement is true. (The case when \( V > U = (M \to N) \) is similar.)

(c) If \((M \to N) = U > V \), then

\[
\text{BSVP}(d, h, \text{lca}(m, n), s) = \text{BSVP}(\triangleright d, hba_{U_1}, \text{lca}(m, n), s) \to \\
(\text{BSVP}(\triangleright d, hba_{U_2}, \text{lca}(m, n), s) \circ \text{BSVP}(\triangleright d, hba_{V}, \text{lca}(m, n), s)).
\]

Also,

\[
\text{BSVP}(d, h, m, s) = \text{BSVP}(\triangleright d, hba_{U_1}, m, s)
\]

and

\[
\text{BSVP}(d, h, n, s) = \text{BSVP}(\triangleright d, hba_{U_2}, n, s) \circ \text{BSVP}(\triangleright d, hba_{V}, n, s),
\]

where

\[
\text{BSVP}(\triangleright d, hba_{V}, \text{lca}(m, n), s) = \text{BSVP}(\triangleright d, hba_{V}, n, s)
\]

\[
= \text{BSVP}(\triangleright d, hba_{V_1}, n, s) \to \text{BSVP}(\triangleright d, hba_{V_2}, n, s).
\]

Since \( t \to (t_1 \circ t_2) = (t \to t_1) \circ t_2 \) for any truth-triplets \( t, t_1, \) and \( t_2 \) such that \( t \) is unscarred (i.e., \( t\top = t\bot \)), the statement follows immediately by the induction hypothesis. (The case when \((M \to N) = V > U \) is similar.)
(d) If \( U > (M \rightarrow N) > V \), then assuming \( M > V \) (the other cases being similar), we have that

\[
\text{BSVP}(d, h, \text{lca}(m, n), s) = \text{BSVP}(\triangleright d, hba_U, \text{lca}(m, n), s) \circ \text{BSVP}(\triangleright d, hba_V, \text{lca}(m, n), s).
\]

Also,

\[
\text{BSVP}(d, h, m, s) = \text{BSVP}(\triangleright d, hba_U, m, s) \circ \text{BSVP}(\triangleright d, hba_V, m, s)
\]

and

\[
\text{BSVP}(d, h, n, s) = \text{BSVP}(\triangleright d, hba_U, n, s),
\]

where

\[
\text{BSVP}(\triangleright d, hba_V, \text{lca}(m, n), s) = \text{BSVP}(\triangleright d, hba_V, m, s) = \text{BSVP}(\triangleright d, hba_V, 1, m, s) \rightarrow \text{BSVP}(\triangleright d, hba_V, 2, m, s).
\]

Since \((t_1 \circ t_2) \rightarrow t = (t_1 \rightarrow t) \circ t_2\) for any truth-triplets \(t_1\), \(t_1\), and \(t_2\) such that \(t\) is unscarred, the statement follows immediately by the induction hypothesis. (The case when \( U > (M \rightarrow N) > V \) is similar.)

(e) The case when \((M \rightarrow N) > U, V\) is similar to the last one, and all cases can be suitably simplified when \( B > U \) or \( B > V \).

2. We will actually prove that for all supersentences \(s\) of \(x\), given a mask \(m\) for \(x\) in \(s\) and a suitable value for \(d\), \(\text{BSVP}(d, \varepsilon, m, s) = \text{BSVP}(d_x, \varepsilon, m_x, s_x)\), where \(s_x, m_x, \text{ and } d_x\) are the default values for \(x\). This will be proved by induction on the number of leaves of the sentence encoded by \(x\). If \(x\) encodes a sentence with only one leaf, then \(x\) either encodes “\(\top\)” or “\(\bot\)”, in which case there is a unique history \(h\) that picks out \(x\) from any given sentence \(s\) containing \(x\). For that history, it is easy to see that \(\text{BSVP}(d, \varepsilon, m, s) = \text{BSVP}(1, h, m, s)\) which is equal to the value of \(x\), and the same is true for the default values of \(s, m, \text{ and } d\).

Now, suppose that \(x\) encodes a sentence \(M' = (M \rightarrow N)\) with \(k > 1\) leaves, and let \(s\) be any supersentence of \(x\). Then,

\[
\text{BSVP}(d, \varepsilon, m', s) = \text{BSVP}(d, \varepsilon, m, s) \rightarrow \text{BSVP}(d, \varepsilon, n, s)
\]

\[
= \text{BSVP}(d_x, \varepsilon, m_x, s_x) \rightarrow \text{BSVP}(d_x, \varepsilon, n_x, s_x)
\]

\[
= \text{BSVP}(d_x, \varepsilon, m'_x, s_x)
\]

where the first and last equality are true by Claim 5.3.1[[1]], and the middle equality is true by the induction hypothesis. □
5.4 The soundness proof

5.4.1 Preliminaries

If we let $\text{TRUE}(v, x, z) = \text{formula}(x, z) \land^B \text{VALUE}(v, x, z)$, then we can express the fact that our $F$-system is sound with the following statement in $T_1$: $\text{TRUE}(v, F(x, y, z, w), z) = 1$.

We will show that $T_1$ can prove this statement, by induction on the parameter $w$ (which indicates the number of lines in the proof encoded by $x, y$). For this, though, we will have to define a form of “strong induction” in $T_1$. First, we need to define a notion of prefix for strings: “$y \sqsubseteq x$”, defined below, returns 1 when $y$ is a prefix of $x$, 0 otherwise.

$$y \sqsubseteq x = y =^S \text{lct}(x, y).$$

Next, we will formalize the notion of “part-of” quantifiers in $T_1$. More precisely, we will show how to represent the part-of quantifications $\bigwedge_{y \sqsubseteq x} A[y]$ and $\bigvee_{y \sqsubseteq x} A[y]$ for any fixed formula $A$. Since we have shown that $=^S$ is equivalent to $=$ in $T_1$ and that the connectives of $T_1$ are equivalent to their functional counterparts, we can replace $=$ with $=^S$, $\neg$ with $\neg^B$, etc. inside $A[y]$ to obtain a term $\hat{A}(y)$ for which we know $T_1$ can prove $A[y] \leftrightarrow \hat{A}(y) = 1$, and this for each value of $y$. Then, if we define

$$\text{cat}_A(y) = \text{cat}_A(y) \cdot \hat{A}(yi)$$

by CRN, it is immediately clear that

$$\bigwedge_{y \sqsubseteq x} A[y] \leftrightarrow \text{AND}(\text{cat}_A(x)) = 1,$$

$$\bigvee_{y \sqsubseteq x} A[y] \leftrightarrow \text{OR}(\text{cat}_A(x)) = 1.$$

Now, suppose that for a particular formula $A$, we can prove in $T_1$ that $A[\varepsilon]$ and $\bigwedge_{y \sqsubseteq x} A[y] \rightarrow A[xi]$. Can we conclude that $A$ is true? An easy NIND on $x$ proves $\bigwedge_{y \sqsubseteq x} A[y]$ in $T_1$:

1. $\bigwedge_{y \sqsubseteq \varepsilon} A[y] = A[\varepsilon]$, which we can prove by assumption;

2. assuming that $\bigwedge_{y \sqsubseteq x} A[y]$, an application of modus ponens gives us $A[xi]$, which implies that $\bigwedge_{y \sqsubseteq xi} A[y]$ holds by properties of AND and the definition of $\text{cat}_A$.

Hence, $T_1$ proves $\bigwedge_{y \sqsubseteq x} A[y]$, which implies, in particular, that $A[x]$ holds.

5.4.2 The proof

**Theorem 5.4.1** $T_1$ proves $\text{TRUE}(v, F(x, y, z, w), z) = 1$. 
We use the “strong induction” described above to prove the theorem. The proof itself will be quite short.

First, \( F(x, y, z, \varepsilon) = \#_{|\text{pow}^L(z)|}^T \), and \( \text{TRUE}(v, \#_{|\text{pow}^L(z)|}^T, z) \) is obviously equal to 1 by Theorem 5.3.1. From now on, we will implicitly assume that \( \text{ALOGTIME} \) of \( d \),

We use the “strong induction” described above to prove the theorem. The proof itself will be quite short.

First, \( F(x, y, z, \varepsilon) = \#_{|\text{pow}^L(z)|}^T \), and \( \text{TRUE}(v, \#_{|\text{pow}^L(z)|}^T, z) \) is obviously equal to 1 by Theorem 5.3.1. From now on, we will implicitly assume that \( F(x, y, z, w) \) is not equal to \( \#_{|\text{pow}^L(z)|}^T \).

Next, assume that \( \bigwedge_{u \subseteq w} \text{TRUE}(v, F(x, y, z, u), z) = 1 \), and consider the following cases, based on the value of \( y_{|w|} \), for the value of \( \text{TRUE}(v, F(x, y, z, w1), z) \).

- If \( y_{|w|} = (0, 1) \), then \( F(x, y, z, w1) \) is the encoding of a formula of the form \( (A \rightarrow (B \rightarrow A)) \), in which case Theorem 5.3.1 implies that

\[
\text{TRUE}(v, F(x, y, z, w1), z) = \text{TRUE}(v, \#_{|\text{pow}^L(z)|}^T A, z) \rightarrow B (\text{TRUE}(v, \#_{|\text{pow}^L(z)|}^T B, z) \rightarrow B \text{TRUE}(v, \#_{|\text{pow}^L(z)|}^T A, z)),
\]

which is obviously equal to 1 in \( T_1 \), being a simple tautology.

- The other three axioms can easily be dealt with similarly.

- If \( y_{|w|} = (k_1, k_2) \), then we know that \( F(x, y, z, \tilde{k_2}) = \pi_{k_2}^{|w|}(x) \) encodes a formula \( A_{k_2} = A_{k_1} \rightarrow A_{|w|} \), where \( A_{k_1} \) is the formula encoded by \( F(x, y, z, \tilde{k_1}) = \pi_{k_1}^{|w|}(x) \) and \( A_{|w|} \) is the formula encoded by \( F(x, y, z, w1) \). But then, by the induction hypothesis, we know that \( \text{TRUE}(v, F(x, y, z, \tilde{k_1}), z) = 1 \) and

\[
\text{TRUE}(v, F(x, y, z, \tilde{k_2}), z) = \text{TRUE}(v, F(x, y, z, \tilde{k_1}), z) \rightarrow B \text{TRUE}(v, F(x, y, z, w1), z) = 1,
\]

so it immediately follows that \( \text{TRUE}(v, F(x, y, z, w1), z) = 1 \).

### 5.5 Simulation results

In this section, we show that \( \mathcal{F} \) can \( p \)-simulate any proof system \( S \) whose soundness can be proved in \( T_1 \). A similar result was first proved for \( PV \) by Cook [18] (Krajíček gives a more detailed proof in his book [23, Theorem 9.3.17]), but to our knowledge, this is the first theory of \( ALOGTIME \) reasoning for which such a result is shown.

Intuitively, the proof hinges on the fact that for any formula \( A \), the propositional translations of the \( T_1 \) equation “\( \text{TRUE}(v, \#_{2^p} A, l) = 1 \)” can be proven equivalent to a substitution instance of \( A \) itself, so that if \( T_1 \) proves the equation, then \( A \) is a tautology with short \( \mathcal{F} \)-proofs.

More precisely, recall from Chapter 4 that for any equation \( t = u \) of \( T_1 \), we defined a family of propositional translations \( [t = u]^{\tilde{m}, \tilde{n}} \) that have polysize \( \mathcal{F} \)-proofs whenever \( t = u \) is a theorem of \( T_1 \). Hence, for any formula \( A \) and string \( l \) such that \( |l| = 2^\ell \geq \ell_A + 2 \), if \( T_1 \) proves
5.5. Simulation results

\( \text{TRUE}(v, \#_p^2 A, L) = 1 \), then there are polysize \( F \)-proofs of the corresponding propositional tautologies \( \{ \text{TRUE}(v, \#_p^2 A, L) = 1 \}^k \) (where \( k \) is the length of the free variable \( v \)), and these tautologies are defined as \( \langle \text{TRUE}(v, \#_p^2 A, L) \rangle^k_i \leftrightarrow \top \), which is equivalent to \( \langle \text{TRUE}(v, \#_p^2 A, L) \rangle^k_1 \) (the term formula describing the first and only bit of the term \( \text{TRUE}(v, \#_p^2 A, L) \) as a function of the bits of its free variable \( v \)). Also, since \( T_1 \) can prove \( \text{TRUE}(v, \#_p^2 (A \rightarrow B), L) = \text{TRUE}(v, \#_p^2 A, L) \rightarrow_b^b \text{TRUE}(v, \#_p^2 B, L) \) for any formulas \( A \) and \( B \), there are polysize \( F \)-proofs that the corresponding propositional translations are equivalent, i.e.,

\[
\langle \text{TRUE}(v, \#_p^2 (A \rightarrow B), L) \rangle^k_1 \leftrightarrow \left( \langle \text{TRUE}(v, \#_p^2 A, L) \rangle^k_1 \rightarrow \langle \text{TRUE}(v, \#_p^2 B, L) \rangle^k_1 \right),
\]

so that if \( T_1 \) proves \( \text{TRUE}(v, \#_p^2 (A \rightarrow B), L) = 1 \), then there are polysize \( F \)-proofs of

\[
\langle \text{TRUE}(v, \#_p^2 A, L) \rangle^k_1 \rightarrow \langle \text{TRUE}(v, \#_p^2 B, L) \rangle^k_1.
\]

Applying this reasoning recursively shows that if \( T_1 \) proves \( \text{TRUE}(v, \#_p^2 A, L) = 1 \), then there are polysize \( F \)-proofs (call them \( \Pi_k \)) of \( \mathcal{A} \{ \langle \text{TRUE}(v, \#_p^2 p_i, L) \rangle^k_1 / p_i \} \) (i.e., the formula \( \mathcal{A} \) where each propositional variable \( p_i \) has been replaced by the formula \( \langle \text{TRUE}(v, \#_p^2 p_i, L) \rangle^k_1 \)). Since the subformulas \( \langle \text{TRUE}(v, \#_p^2 p_i, L) \rangle^k_1 \) are never “broken up” inside \( \Pi_k \), we can just substitute \( p_i \) for \( \langle \text{TRUE}(v, \#_p^2 p_i, L) \rangle^k_1 \) throughout to get polysize \( F \)-proofs of \( \mathcal{A} \).

Now, since \( \mathcal{F} \) can “evaluate” sentences (i.e., given a propositional formula with no variables, \( \mathcal{F} \) has polysize proofs that it is equivalent to its truth-value), for any function \( f(x_1, \ldots, x_n) \) definable in \( T_1 \), and any tuple of strings \( a_1 \in \{0,1\}^*, \ldots, a_n \in \{0,1\}^* \), there are polysize \( \mathcal{F} \)-proofs that \( \langle f(\vec{x}) \rangle^m_i \left[ \langle a_j \rangle_{i_j} / \langle x_j \rangle^{m_j}_{i_j} \right] \leftrightarrow \langle f(\vec{a}) \rangle_i \) (where \( \langle a_j \rangle_{i_j} \) is equal to \( \top \) or \( \bot \) depending on the value of bit number \( i \) of \( a_j \)). In particular, if \( S \) is a proof system formalizable in \( T_1 \) as a function symbol \( S(x, z) \), then for any particular \( S \)-proof \((a, b)\) of a formula \( \mathcal{A} \), there are polysize \( \mathcal{F} \)-proofs that \( S(a, b) \) is equivalent to the encoding of \( \mathcal{A} \).

Putting these two facts together, we have that if \( S(x, z) \) is a proof system whose soundness can be proved in \( T_1 \) (i.e., for which \( T_1 \) can prove \( \text{TRUE}(v, S(x, z), z) = 1 \)), then for any particular \( S \)-proof \((a, b)\), there are polysize \( \mathcal{F} \)-proofs of the formula encoded by \( S(a, b) \), i.e., \( \mathcal{F} \) \( p \)-simulates \( S \). Moreover, it appears that the translation from \( S \)-proofs to \( \mathcal{F} \)-proofs can be carried out in \( NC^1 \) and thus formalized in \( T_1 \), where the simulation proof can also be formalized (although we do not carry this out, we do not expect any technical difficulties to arise in the details of such a formalization).

Remark 5.5.1 Note that Theorems 4.4.1 and 5.4.1 immediately give an alternative proof that \( \mathcal{F} \)-systems have polysize proofs of their own partial consistency (when suitably expressed), a fact first proved directly by Buss [9]. The partial self-consistency statements obtained through \( T_1 \) would be different from the ones Buss considered, but it should be possible to prove that they are equivalent.
Chapter 6

Related Work

In this chapter, we show that Arai’s AID is equivalent to QT$_1$ (a suitably defined quantified version of T$_1$), and briefly discuss the relationship between T$_1$ and Clote’s ALV or ALV$'$. We will keep the discussion at a high level, with few technical details.

6.1 T$_1$ and AID

If we define QT$_1$ to be a first-order theory whose non-logical symbols are those of T$_1$ and whose axioms are the universal closures of the axioms of T$_1$, together with axiom schemes corresponding to NIND and TIND, then we can show that

- QT$_1$ is a conservative extension of T$_1$ (the proof is similar to Cook’s proof in [17] that QALV is conservative over ALV$'$);

- for every $\Sigma^0_0$-formula $B$ in QT$_1$, there exists a function symbol $\hat{B}$ in T$_1$ such that QT$_1$ proves $B[\bar{x}] \rightarrow \hat{B}(\bar{x}) = 1$ (sharply bounded quantifiers, e.g. “$\forall x \leq |t|$”, are easy to represent functionally since we already have “part-of” quantifiers from the end of Chapter 5 and $\forall x \leq |t|B[x] \leftrightarrow \bigwedge_{x \subseteq t} B[|x|]$, for example);

- QT$_1$ proves the scheme of $\Sigma^0_0$-LIND (with a straightforward application of NIND).

Next, the primitive functions of AID are all easily defined in T$_1$ (all treating their inputs “numerically”, i.e., ignoring leading zeroes), and their defining axioms can be proven without difficulty. Also, for every inductively defined predicate $A^{t,B,\vec{D},I}$ in AID, we can define a $\{0,1\}$-valued function $\hat{A}$ in T$_1$ such that the equation $\hat{A}(\bar{x},p) = 1$ provably satisfies the defining axioms (A.0)–(A.2) of $A$. This can be done by TRN in a relatively straightforward manner, except for two technicalities that we discuss now.

First, the recursive definition of $A$ in AID involves computing the values of predicates $D_1(\bar{x},p), \ldots, D_m(\bar{x},p)$ at every level of the recursion, even though these computations can only
be represented in $T_1$ by function symbols of rank 1 (i.e., the computations are in $NC^1$). Since functions of rank 1 are not allowed in the recursive part of a definition by TRN, we need to precompute the values of the $\vec{D}$ predicates for every level and extract the correct values during the recursive definition by TRN. This is accomplished by first computing the concatenation of all the values of $p$ for which the $\vec{D}$ predicates need to be computed, then using CRN to compute the concatenation of the $\vec{D}$ predicates for each value, and finally breaking up this string in the appropriate way during the recursion (so the order in which the values are listed is important and must be chosen carefully).

Second, the depth of the recursion is controlled by the “linear form” $\ell||\vec{x}||$, which is represented in $T_1$ by a numerical function, i.e., one whose actual string length could be arbitrarily longer than its numerical length because of leading 0’s. By simply extending the precomputation carried out above for the $\vec{D}$ predicates so that the values of the $B$ predicate are also computed for each value of $p$ at the “bottom” of the recursion, we can easily define the term $\hat{A}$ by TRN so that the recursion terminates early once the base cases are reached, so that the exact string length of the parameter controlling the TRN does not matter, as long as it is long enough.

In the other direction, every primitive function of $T_1$ can easily be defined in $AID$, using Arai’s representation of strings (a string $b_{k-1} \cdots b_0$ is represented by the integer $1b_{k-1} \cdots b_0$), since the primitive “part” function of $AID$ can be used directly to extract arbitrary substrings. From the properties of “part”, the defining axioms for the $T_1$-functions can be proven without difficulty in $AID$. Also, function symbols defined by $\ell$CRN or $r$CRN in $T_1$ can be defined in $AID$ in a straightforward manner using Comprehension, and the defining axioms for these functions follows directly from the Comprehension axiom in $AID$. Finally, functions defined by TRN in $T_1$ can be defined in $AID$ using an inductively defined predicate $A_{\ell,B,\vec{D},I}$ that uses its parameter $p$ to keep track of a path through the recursion tree and that computes the appropriate substring of $x$ and the appropriate function of $z$ at each level using the predicates $\vec{D}$. Moreover, the defining axioms of such a function symbol follow directly from the axioms for $A$ in $AID$.

Now, for every formula $B$ in $AID$, we let $\hat{B}$ denote the formula in $QT_1$ obtained from $B$ by replacing each primitive function symbol by its definition in $T_1$ and each inductively defined predicate symbol $A$ by its definition $\hat{A}$ in $T_1$. Similarly, for every formula $B$ in $QT_1$, we let $\tilde{B}$ denote the formula in $AID$ obtained from $B$ by replacing each function symbol of $T_1$ by its definition in $AID$.

These translations allow us to show that if a formula $B$ is provable in $AID + \Sigma^b_0$-CA, then $QT_1$ can prove $\hat{B}$ (since $QT_1$ proves $\Sigma^b_0$-LIND and $\Sigma^b_0$-CA can be defined using CRN and proven using NIND), and that if a formula $B$ is provable in $QT_1$, then $AID + \Sigma^b_0$-CA can prove $\tilde{B}$ (since $AID + \Sigma^b_0$-CA can prove NIND using Comprehension, and TIND in a way similar to
6.2. \( T_1 \) and \( PV \)

Based on a similar result of Buss [11], which uses Krajíček, Pudlák & Takeuti’s Herbrand-type witnessing theorem [23], Cook [17] has argued that if \( QPV \) is conservative over \( QALV \), then \( P = ALOGTIME \), where \( QPV \) is the appropriately defined quantified theory corresponding to \( PV \) and \( QALV \) is a quantified theory based on Clote’s \( ALV' \). A similar result should hold with \( QT_1 \) in place of \( QALV \), given a suitable interpretation of strings as numbers and numbers as strings.

As for the quantifier-free theories \( PV \) and \( T_1 \), it is easy to see that if \( PV \) is conservative over \( T_1 \) (through an appropriate translation between numbers and strings, such as the one used above), then \( T_1 \) proves the soundness of \( eF \) (since \( PV \) proves the soundness of \( eF \) and \( eF \) can be defined in \( T_1 \)), so that \( F \) \( p \)-simulates \( eF \). Unfortunately, the converse is not known, and this has no known implication for the complexity classes \( P \) and \( NC^1 \).
Chapter 7

Conclusion

7.1 Summary

As Chapter 2 has shown, $L_1$ is an elegant and natural recursive characterization of $NC^1$: simple functions are easy to define and even for more complex functions, the definitions are not unnecessarily complicated. The only exception to that statement might be the “numerical” functions, but even there, the definitions are straightforward and correspond quite closely to the computation of these functions by circuit families. Also, our scheme of TRN seems to capture the computational power of $ALOGTIME$ in the most natural way, as evidenced by the short proof that $FALOGTIME \subseteq L_1$. It would be interesting to prove the other direction also (that $L_1 \subseteq FALOGTIME$) by using computations by ATMs as opposed to uniform circuit families, so that both directions of the proof are similar, but time constraints did not allow us to carry out such a proof.

The theory $T_1$ based on $L_1$ has the desirable property that its appropriately translated theorems have short $F$-proofs, and the proof of that fact is quite simple (especially when compared to the corresponding proofs for other theories of $ALOGTIME$ reasoning in the literature). In fact, the structure of the $F$-proofs is straightforward enough that we get precise estimates on their size (as a function of the lengths of the variables). Also, considering the inherent complexity of evaluating Boolean sentences in $ALOGTIME$, the $T_1$-proof of the soundness of a particular $F$-system is straightforward, consisting mainly in the formalization of Buss’s BSVP algorithm and the proof of its properties. Finally, the fact that $F$ $p$-simulates any proof system $S$ whose soundness can be proved in $T_1$ is also straightforward to prove, and $T_1$ is the first theory of $ALOGTIME$ reasoning for which this result has been shown. All these facts strongly support our claim that $T_1$ is one of the most natural theories available for $ALOGTIME$ reasoning, even though it is based on strings instead of numbers (unlike most of the other theories for polytime or $ALOGTIME$ reasoning).
To conclude, it might seem that any algebraic characterization of a complexity class could be used to define a quantifier-free theory like $T_1$, by simply having function symbols defined recursively and induction rules based on the recursion operators. Although such a theory would undoubtedly reason on functions in the desired complexity class, we would still need to show that the type of reasoning that can be carried out in this theory also falls within the desired complexity class, and there is no clear way of doing this for arbitrary complexity classes. Also, as evidenced by Clote’s theories $ALV$ and $ALV'$, it is not an easy task to get a theory that is natural and simple enough to be useful in practice.

### 7.2 Future work

First, an obvious generalization of $L_0$ and $L_1$ suggests itself: for $i \geq 1$, let $L_{i+1}$ be the closure of $L_i$ under COMP, CRN, and $\text{TRN}_{L_i}^{L_{i+1}}$, defined recursively. A study of these classes (or of a similar extension for the theory $T_1$) might be interesting. (One fact about $L_i$ which is relatively easy to prove is that it is a subset of the class of functions computable by uniform circuit families of $O(\log^i)$ depth, but it is unknown if this is a proper containment. It might be interesting to try to prove better results, maybe that the $L_i$’s exactly captures these circuit families, or that the union of the $L_i$ hierarchy defines the class of functions computable by uniform circuit families of polylog depth.)

Next, it would be interesting to compare $QT_1$ to Takeuti and Clote’s $TNC^0$, maybe to show that the two first-order theories are equivalent. Also, from Arai’s results on $AID$ and $ALV$, we have concluded in Chapter 6 that $T_1$ and $ALV$ are equivalent, but it is unknown whether or not $ALV'$ is equivalent to $ALV$ (or to $T_1$).

It would also be interesting to see if “tree recursion” and “tree induction” can be adapted to define a similar quantifier-free theory for uniform $TC^0$ reasoning, hopefully as natural as $T_1$ is natural for $ALOGTIME$ reasoning (such a theory would correspond to bounded-depth $\mathcal{F}$-systems with threshold gates in the same way that $T_1$ corresponds to $\mathcal{F}$-systems). Note that there is already a first-order theory $\tilde{R}_2^0$ for $TC^0$ reasoning, defined by Johannsen [22].

Similarly, there should be a way to extend $L_0$ and $L_1$ to capture all of $NC$ (based on Bloch’s characterization of $NC$). This could possibly be used to define a quantifier-free theory for $NC$ reasoning, which might lead to a natural propositional proof system that reasons in $NC$.

Finally, fully relating conservativity results between logical theories for $P$ and $ALOGTIME$ reasoning to equivalence results between $e\mathcal{F}$ and $\mathcal{F}$ systems to collapse results between $P$ and $NC^1$ remains a central open problem in this area.
Appendix A

Details of Proofs in the Formal Development of $T_1$

This appendix contains most of the proofs missing from the formal development of $T_1$ given in Chapter 3. It is included here mainly for the sake of completeness, so the style will be quite terse. In particular, most proofs that consist only in a straightforward application of NIND will be omitted.

On generalizations of NIND

**Claim 3.2.54**

1. (L) $z \triangleright yx = z \triangleright y \cdot (z \triangleleft y) \triangleright x$  
   (R) $xy \triangleleft z = x \triangleleft (y \triangleright z) \cdot y \triangleleft z$

2. (L) $y \triangleleft x = lb(x, y) \cdot y \triangleright x$  
   (R) $x \triangleleft \triangleright y = x \triangleleft y \cdot rb(x, y)$

3. (L) $y \triangleright ((x \triangleleft y) \triangleright x) = \varepsilon$  
   (R) $(x \triangleleft (y \triangleright x)) \triangleleft y = \varepsilon$

4. (L) $lc(rc(x, y), y) = rc(x, y)$  
   (R) $rc(lc(x, y), y) = lc(x, y)$

5. (L) $(x \triangleright y) = x \triangleright y ?^{2L} (\varepsilon, (x \triangleleft \triangleright (y \triangleright x))')$  
   (R) $(x \triangleleft y)' = x \triangleleft y ?^{2L} (\varepsilon, ((x \triangleleft y) \triangleright \triangleleft))$  

6. (L) $lc(x, yi) = lc(x, y) \cdot lb(x, yi)$  
   (R) $rc(x, iy) = rb(x, iy) \cdot rc(x, y)$

7. (L) $lc(x, y) \cdot y \triangleright x = x$  
   (R) $x = x \triangleleft y \cdot rc(x, y)$

**Proof**

1. (L) By NIND on $z$, Axioms [4c] and [5c] and Claim 3.2.32: $\varepsilon \triangleright yx = yx = \varepsilon \triangleright y \cdot \varepsilon \triangleright x = \varepsilon \triangleright y \cdot (\varepsilon \triangleleft y) \triangleright x$, $iz \triangleright yx = \triangleright(z \triangleright yx) = \triangleright(z \triangleright y \cdot (z \triangleleft y) \triangleright x) = z \triangleright y ?^{2L} (\triangleright((z \triangleleft y) \triangleright x), \triangleright(z \triangleright y) \cdot (z \triangleleft y) \triangleright x) = z \triangleright y ?^{2L} (\varepsilon \cdot (i \cdot z \triangleleft y) \triangleright x, iz \triangleright y \cdot \varepsilon \triangleright x) = z \triangleright y ?^{2L} (iz \triangleright y \cdot (iz \triangleleft y) \triangleright x, iz \triangleright y \cdot (iz \triangleleft y) \triangleright x) = iz \triangleright y \cdot (iz \triangleleft y) \triangleright x$.  

109
2. (L) By NIND on $y$: $\varepsilon \triangleleft x = x = \varepsilon \cdot x = \text{lb}(x, \varepsilon) \cdot \varepsilon \triangleright x$, $(yi) \triangleleft y = x = \neg(y \triangleright x) \cdot (yi) \triangleright x = \neg((yi) \triangleleft x) \cdot ((yi) \triangleright x) = \text{lb}(x, yi) \cdot (yi) \triangleright x$.

3. (L) By NIND on $y$, and preceding claims: $\varepsilon \triangleright ((x \triangleleft y) \triangleright x) = x \triangleright x = \varepsilon$,

$$
yi \triangleright ((x \triangleleft yi) \triangleright x) = \neg(y \triangleright ((x \triangleleft yi) \triangleright x))
= y \triangleright (\neg(y \triangleright ((x \triangleleft yi) \triangleright x)) \cdot (x \triangleleft y) \triangleright x)
= y \triangleright ((x \triangleleft y) \triangleright x) = \varepsilon
$$

(Note that in this proof, we did not explicitly deal with the cases when $x \triangleleft y = \varepsilon$ or when $(x \triangleleft y) \triangleleft x = \varepsilon$. However, it can easily be seen that both cases make the statement trivially true.)

4. (L) By preceding claims: $\text{lc}(rc(x, y), y) = ((x \triangleleft y) \triangleright x) \triangleright (y \triangleright ((x \triangleleft y) \triangleright x)) = rc(x, y) \triangleleft \varepsilon = rc(x, y)$.

5. (L) By NIND on $x$ and preceding claims: $\neg(y \triangleright \varepsilon) = \varepsilon = y \triangleright \varepsilon \triangleleft 2L (\varepsilon, \varepsilon), \neg(y \triangleright xi) = y \triangleright xi \triangleleft 2L (\varepsilon, (y \triangleright x \cdot i)) = y \triangleright xi \triangleleft 2L (\varepsilon, y \triangleright x \triangleleft 2L (i, (x \triangleleft (y \triangleright x))^i)) = y \triangleright xi \triangleleft 2L (\varepsilon, (x \triangleleft (y \triangleright x))^i) = y \triangleright xi \triangleleft 2L (\varepsilon, (x \triangleleft (y \triangleright xi))^i)$.

6. (L) By preceding claims: $\text{lc}(x, yi) = x \triangleleft (yi \triangleright x) = x \triangleleft (y \triangleright x) = x \triangleleft (y \triangleright x) \cdot y \triangleright x \triangleleft 2L (\varepsilon, (x \triangleleft (y \triangleright x))^i) = x \triangleleft (y \triangleright x) \cdot \neg(y \triangleright x) = \text{lc}(x, y) \cdot \text{lb}(x, yi)$. □

**On propositional reasoning**

**Theorem 3.2.6**

1. $\approx^0 x = 1 \lor \approx^0 x = 0$
2. $\approx^0 \approx^0 x = \approx^0 x$
3. $\neg \approx^0 x = 1 \leftrightarrow \neg(\approx^0 x = 1)$
4. $x \land \approx^0 y = 1 \leftrightarrow (\approx^0 x = 1 \land \approx^0 y = 1)$
5. $x \lor \approx^0 y = 1 \leftrightarrow (\approx^0 x = 1 \lor \approx^0 y = 1)$
6. $x \rightarrow \approx^0 y = 1 \leftrightarrow (\approx^0 x = 1 \rightarrow \approx^0 y = 1)$
7. $x \leftarrow \approx^0 y = 1 \leftrightarrow (\approx^0 x = 1 \leftrightarrow \approx^0 y = 1)$
8. $x \oplus \approx^0 y = 1 \leftrightarrow (\approx^0 x = 1 \oplus \approx^0 y = 1)$

**Proof** The first three can be proved by straightforward NIND on $x$, all the others with a simple and direct application of Derived Rule 3.2.3. □
On “AND” and “OR”

**Lemma 3.2.56**

1. (L) \( x \neq \varepsilon \wedge x \neq 0 \wedge x \neq 1 \rightarrow (\rightharpoonup x \cdot j) \downarrow = (x \downarrow) \cdot \lrcorner x \)

   (R) \( x \neq \varepsilon \wedge x \neq 0 \wedge x \neq 1 \rightarrow (\rightharpoonup x \cdot j) = \rightharpoonup x \cdot j \)

2. (L) \( x \wedge^B \text{AND}(\rightharpoonup x \cdot j) = \text{AND}(x) \wedge^B j \)

   (R) \( j \wedge^B \text{AND}(x) = \text{AND}(j \cdot x) \wedge^B x' \)

3. \( x \wedge^B \text{AND}(\rightharpoonup x) = \text{AND}(x) = \text{AND}(x \lhd) \wedge^B x' \) for \( x \neq \varepsilon, 0, 1 \)

**Proof**

1. (L) By Derived Rule 3.2.11 since \( \varepsilon = \varepsilon \), the statement is trivially true for \( x = \varepsilon \). Assuming \( ix \neq 0 \wedge ix \neq 1 \), i.e., \( x \neq \varepsilon \),

   \[
   (\rightharpoonup (ix) \cdot j) \downarrow = (xj) \downarrow
   \]

   \[
   = x ? \text{EL}(x \downarrow, x \downarrow \cdot \lrcorner (x \cdot j))
   \]

   \[
   = x ? \text{EL}(x \downarrow, x \downarrow \cdot \lrcorner x)
   \]

   \[
   = x ? \text{EL}(x \downarrow < \cdot x \downarrow', (i \cdot x) \downarrow \cdot \lrcorner x)
   \]

   \[
   = x ? \text{EL}(i \cdot x \downarrow \cdot ((i \cdot x)') \cdot \lrcorner x), (i \cdot x) \downarrow \cdot \lrcorner x
   \]

   \[
   = \rightharpoonup((ix) \downarrow) \cdot \lrcorner (ix).
   \]

   (A similar proof shows the same theorem with \( i \cdot x \lhd \) in place of \( \rightharpoonup x \cdot i \).)

2. (L) By the preceding lemma and by TIND on \( x \): \( \lhd \varepsilon \wedge^B \text{AND}(\rightharpoonup \varepsilon \cdot j) = 0 = \text{AND}(\varepsilon) \wedge^B j \), \( \lhd i \wedge^B \text{AND}(\rightharpoonup i \cdot j) = i \wedge^B \text{AND}(j) = \text{AND}(i) \wedge^B j \), and assuming that \( \lhd (x \downarrow) \wedge^B \text{AND}(\rightharpoonup (x \downarrow) \cdot j) = \text{AND}(x \downarrow) \wedge^B j \) and \( \lhd (x \lhd) \wedge^B \text{AND}(\rightharpoonup (x \lhd) \cdot j) = \text{AND}(x \lhd) \wedge^B j \), then, for \( x \) such that \( \rightharpoonup x \neq \varepsilon \) (the case when it is equal being trivial),

   \[
   \lhd x \wedge^B \text{AND}(\rightharpoonup x \cdot j) = \lhd x \wedge^B \text{AND}((\rightharpoonup x \cdot j) \downarrow) \wedge^B \text{AND}(\rightharpoonup (x \cdot j))
   \]

   \[
   = \lhd (x \downarrow) \wedge^B \text{AND}(\rightharpoonup (x \downarrow) \cdot \lrcorner x) \wedge^B \text{AND}(\rightharpoonup x \cdot j)
   \]

   \[
   = \text{AND}(x \downarrow) \wedge^B \text{AND}(\rightharpoonup x \cdot j)
   \]

   \[
   = \text{AND}(x \downarrow) \wedge^B \text{AND}(\rightharpoonup x) \wedge^B j
   \]

   \[
   = \text{AND}(x) \wedge^B j.
   \]

3. By preceding lemmas and by Derived Rule 3.2.5 on \( x \): \( \lhd (ij) \wedge^B \text{AND}(\rightharpoonup (ij)) = i \wedge^B \text{AND}(j) = \text{AND}(ij) = \text{AND}(i) \wedge^B j = \text{AND}((ij) \lhd) \wedge^B (ij) \), \( \lhd (ik) \wedge^B \text{AND}(\rightharpoonup (ik)) = i \wedge^B \text{AND}(kj) = \text{AND}(ik) = \text{AND}(ik) \wedge^B j = \text{AND}((ik) \lhd) \wedge^B (ikj) \), and under the induction hypotheses
that \(\forall (x\triangleup)\wedge^B \text{AND}(\succ (x\triangleup)) = \text{AND}(x) = \text{AND}((x\triangleup)\prec^B(x\triangleup)'\) and \(\forall (\triangledown x)\wedge^B \text{AND}(\succeq (\triangledown x)) = \text{AND}(x) = \text{AND}((\triangledown x)\prec^B(\triangledown x)')\) for \(x\triangleup >^L 1\), then

\[
\forall x \wedge^B \text{AND}(\succ x) = \forall x \wedge^B \text{AND}((\succ x)\triangleup) \wedge^B \text{AND}((\triangledown x))
\]

\[
= x \ ?^\text{EL} (\forall (x\triangleup) \wedge^B \text{AND}(\succ (x\triangleup))) \wedge^B \text{AND}(\triangledown x),\]

\[
\forall (x\triangleup) \wedge^B \text{AND}(\succ (x\triangleup)) \wedge^B \text{AND}(\triangledown x))
\]

\[
= x \ ?^\text{EL} (\text{AND}(x\triangleup) \wedge^B \text{AND}(\triangledown x), \text{AND}(x\triangleup) \wedge^B \text{AND}(\triangledown (\triangledown x)))
\]

\[
= x \ ?^\text{EL} (\text{AND}(x\triangleup) \wedge^B \text{AND}(\triangledown x), \text{AND}(x\triangleup) \wedge^B \text{AND}(\triangledown (\triangledown x)))
\]

\[
= x \ ?^\text{EL} (\text{AND}(x\triangleup) \wedge^B \text{AND}(\triangledown x), \text{AND}(x\triangleup) \wedge^B \text{AND}(\triangledown (\triangledown x)))
\]

On generalizations of CRN—part I

**Theorem 3.2.9**  
\(jx = jy \iff x \triangledown y = \varepsilon = x \prec y\)

**Proof**  
By Derived Rule 3.2.23  
\(j\varepsilon = jy \iff \varepsilon = y \iff \varepsilon \iff y = \varepsilon \iff jx = j\varepsilon \iff x = \varepsilon \iff x \triangledown \varepsilon = \varepsilon = x \prec \varepsilon, j(x\iota) = j(ky) \iff j\cdot jy \iff jx = jy \iff x \triangledown y = \varepsilon = x \prec y \iff x \triangledown ky = \varepsilon = xi \prec ky.\)

**Claim 3.2.57**  
For \(f = r\text{CRN}[h],\)

1. \(j(f(x, \bar{y})) = jx\)

2. \(f(x, \bar{y}) \prec z = f(x \prec z, \bar{y})\)

3. \(\text{lb}(f(x, \bar{y}), z) = x \triangledown z \ ?^\text{EL} ((0 \cdot h(\text{lc}(x, z), \bar{y}))', \varepsilon)\) for \(x, z \neq \varepsilon\)

4. \(\text{lc}(f(x, \bar{y}), z) = f(\text{lc}(x, z), \bar{y})\)

**Proof**  
1. By NIND on \(x:\) \(j(f(\varepsilon, \bar{y})) = j\varepsilon, j(f(xi, \bar{y})) = j(f(x, \bar{y}) \cdot (0 \cdot h(xi, \bar{y}))') = jx \cdot j = j(xi)\).
2. By Derived Rule 3.2.38, \( f(ε, \vec{y}) \preceq z = ε \preceq z = f(ε \preceq z, \vec{y}), f(x, \vec{y}) \preceq ε = f(x, \vec{y}) = f(x \preceq ε, \vec{y}), f(xi, \vec{y}) \preceq zj = (f(x, \vec{y}) \cdot (0 \cdot h(xi, \vec{y}))zj) \preceq z = f(x, \vec{y}) \preceq z = f(x \preceq z, \vec{y}) = f(xi \preceq zj, \vec{y}).

3. First, a straightforward proof by NIND on \( x \) shows that \( f(x, \vec{y}) = (0 \cdot h'(x, \vec{y}))' \). Now, by NIND on \( z \neq ε \) and the claims above: \( lb(f(x, \vec{y}), j) = f(ε \triangleright f(x, \vec{y})) = (0 \cdot h'(x, \vec{y}))' = (0 \cdot h(lc(x, j), \vec{y}))' \),

\[
\begin{align*}
lb(f(x, \vec{y}), zi) &= (z \triangleright f(x, \vec{y})) \\
&= z \triangleright f(x, \vec{y}) \\
&= z \triangleright f(x, \vec{y}) \\
&= z \triangleright f(x, \vec{y}) \\
&= z \triangleright f(x, \vec{y}) \\
&= x \triangleright zi \cdot (0 \cdot h(lc(x, z) \cdot lb(x, zi), \vec{y}))', ε \\
&= x \triangleright zi \cdot (0 \cdot h(lc(x, zi), \vec{y}))', ε.
\end{align*}
\]

4. By NIND on \( z \): \( lc(f(x, \vec{y}), ε) = ε = f(lc(x, ε), \vec{y}) \), and

\[
\begin{align*}
lc(f(x, \vec{y}), zi) &= lc(f(x, \vec{y}), z) \cdot lb(f(x, \vec{y}), zi) \\
&= f(lc(x, z), \vec{y}) \cdot x \triangleright zi \cdot (0 \cdot h(lc(x, zi), \vec{y}))', ε \\
&= z \triangleright x \cdot (0 \cdot h(lc(x, zi), \vec{y}))', ε \\
&= z \triangleright x \cdot (0 \cdot h(lc(x, zi), \vec{y}))', ε \\
&= f(lc(x, zi), \vec{y}).
\end{align*}
\]

**Lemma 3.2.38**

1. \((L)\) \( y \triangleright x = x \triangleright y \cdot (y \triangleright x, ε) \) \hspace{1cm} \((R)\) \( x \preceq y = x \triangleright y \cdot (0 \cdot h(x, y), ε) \)

2. \((L)\) \( (y \preceq (x \triangleright y)) \triangleright x = y \triangleright x \) \hspace{1cm} \((R)\) \( x \preceq ((y \preceq x) \triangleright y) = x \preceq y \)

**Proof**

1. \((L)\) Immediate from the fact that \( x \triangleright y \neq ε \rightarrow y \triangleright x = ε \).

2. \((L)\) By NIND on \( y \): \( (ε \preceq (x \triangleright ε)) \triangleright x = (ε \preceq ε) \triangleright x = ε \triangleright x, \)

\[
(yj \preceq (x \triangleright yj)) \triangleright x = x \triangleright yj \cdot ((yj \preceq ε) \triangleright x, (yj \preceq ((x \triangleright y) \cdot j)) \triangleright x)
\]

\[
= x \triangleright yj \cdot yj \cdot (y \preceq (x \triangleright y)) \triangleright x
\]

\[
= x \triangleright yj \cdot yj \cdot (y \preceq (x \triangleright y) \cdot ε)
\]

\[
= x \triangleright yj \cdot yj \cdot (y \preceq (x \triangleright y) \cdot ε)
\]

\[
\]
Appendix A. Details of Proofs in the Formal Development of $T_1$

Claim 3.2.59

1. (L) $lp_j(x, \max^C(x, y)) = lp_j(x, y)$ \quad (R) $rp_j(x, \max^C(x, y)) = rp_j(x, y)$

2. (L) $lb(lp_0(x, \max^L(xi, yj)), \max^L(xi, yj)) = i$
   \quad (R) $rb(rp_0(ix, \max^L(iy, jy)), \max^L(iy, jy)) = i$

Proof

1. (L): $lp_j(x, \max^C(x, y)) = x \triangleright y \uparrow 2\uparrow \uparrow (j(x \triangle x) \cdot x, j(y \triangle x) \cdot x) = x \triangleright y \uparrow 2\uparrow \uparrow (\varepsilon \cdot x, j(y \triangle x) \cdot x)$

2. (L):

\[
\begin{align*}
lb(lp_0(xi, \max^L(xi, yj)), \max^L(xi, yj)) &= lb(lp_0(x, \max^L(x, y)), i, \max^L(x, y) \cdot x \triangleright y \uparrow 2\uparrow \uparrow (i, j)) \\
&= \lambda(x, y) \cdot x \triangleright y \uparrow 2\uparrow \uparrow (\varepsilon \cdot (x \triangleright yi) \cdot x) \\
&= x \triangleright y \uparrow 2\uparrow \uparrow \lambda(x, y) \\
&= \lambda(x, y) \cdot (\varepsilon \cdot (x \triangleright yi) \cdot x)
\end{align*}
\]

Theorem 3.2.10

\[
\begin{align*}
\ell CRN_m[h](x_1, \ldots, x_m, \vec{y}) &= x_1 \cdot \ldots \cdot x_m \uparrow 2\uparrow \left(\varepsilon, \lambda(h)) \cdot \ell CRN_m[h](\vec{lp}_0(x_1, \max^L_m(\vec{x}_m), \ldots, \vec{lp}_0(x_m, \max^L_m(\vec{x}_m)), \vec{y})) \cdot 0\right) \\
&= \ell CRN_m[h]\left(\vec{lp}_0(x_1, \max^L_m(\vec{x}_m), \ldots, \vec{lp}_0(x_m, \max^L_m(\vec{x}_m)), \vec{y})\right)
\end{align*}
\]

\[
\begin{align*}
r CRN_m[h](x_1, \ldots, x_m, \vec{y}) &= x_1 \cdot \ldots \cdot x_m \uparrow 2\uparrow \left(\varepsilon, r CRN_m[h]\left(\vec{rp}_0(x_1, \max^L_m(\vec{x}_m)), \ldots, \vec{lp}_0(x_m, \max^L_m(\vec{x}_m)), \vec{y}\right) \cdot 0\right) \\
&= \left(0 \cdot h(\vec{rp}_0(x_1, \max^L_m(\vec{x}_m), \ldots, \vec{rp}_0(x_m, \max^L_m(\vec{x}_m)), \vec{y})\right)'
\end{align*}
\]

Proof From the fact that $\max^L_m(\vec{x}_m) = \varepsilon \leftrightarrow x_1 \cdot \ldots \cdot x_m = \varepsilon$ (easily proved by Derived Rule 3.2.3), the theorem follows from Claim 3.2.59 by a straightforward application of Derived Rule 3.2.3 generalized to $m$ variables. \qed
Claim 3.2.60

1. \( \text{and}_m^B(x_1 i_1, \ldots, x_m i_m) = \text{and}_m^B(\vec{x}_m) \cdot \text{and}_m^B(\vec{i}_m) \)

2. \( j x_1 = \cdots = j x_m \wedge j y_1 = \cdots = j y_m \rightarrow \text{and}_m^B(x_1 y_1, \ldots, x_m y_m) = \text{and}_m^B(\vec{x}_m) \cdot \text{and}_m^B(\vec{y}_m) \).

3. \( \text{and}_m^B(\vec{x}_m) \triangleq \text{and}_m^B(x_1 \triangleq, \ldots, x_m \triangleq) \)

4. \( \text{and}_m^B(\vec{x}_m) \triangleright y = \text{and}_m^B(x_1 \triangleright y, \ldots, x_m \triangleright y) \)

Proof

1. By a straightforward application of Derived Rule 3.2.3, generalized to \( m \) variables.

2. Again, by a straightforward application of Derived Rule 3.2.3, generalized to \( m \) variables, together with the previous claim.

3. From the first claim, by a straightforward generalized NIND.

4. Directly from the preceding claim, with a straightforward NIND on \( y \). \( \square \)

Theorem 3.2.11

\( \neg^B \text{AND}(x) = \text{OR}(\neg^B(x)) \) and \( \neg^B \text{OR}(x) = \text{AND}(\neg^B(x)) \) for \( x \neq \varepsilon \)

Proof (We prove only the first statement, the second one being almost identical.) By TIND on \( x \): \( \neg^B \text{AND}(i) = \neg^B i = \text{OR}(\neg^B i) = \text{OR}(\neg^B(i)) \),

\[
\neg^B \text{AND}(x) = \neg^B(\text{AND}(x \triangleright) \wedge \neg^B \text{AND}(\triangleright x))
\]

\[
= \neg^B \text{AND}(x \triangleright) \lor \neg^B \text{AND}(\triangleright x)
\]

\[
= \text{OR}(\neg^B(x \triangleright)) \lor \neg^B \text{OR}(\neg^B(\triangleright x))
\]

\[
= \text{OR}(\neg^B(x \triangleright)) \lor \neg^B \text{OR}(\triangleright \neg^B(x)) = \text{OR}(\neg^B(x)). \quad \square
\]

On generalizations of CRN—part II

Claim 3.2.61

1. \( \triangleright \text{pow}^L(y) = \text{pow}^L(\triangleright y) = \text{pow}^L(y) \triangleright \) (for \( y \neq \varepsilon, 0, 1 \))

2. \( \downarrow \text{pow}^L(y) = \text{pow}^L(\downarrow y) = \text{pow}^L(y) \)

3. \( \text{pow}^L(\text{pow}^L(y)) = \text{pow}^L(y) \)

4. \( \text{pow}^L(y) \triangleright y = \varepsilon \)

5. \( \text{ispow}^L(\text{pow}^L(y)) = \varepsilon \)
Proof. All can be proved by very simple applications of TIND. We give the proof of the third statement as an illustration: $\text{pow}^L(\text{pow}^L(x)) = \text{pow}^L(x)$, $\text{pow}^L(\text{pow}^L(\epsilon)) = \text{pow}^L(\epsilon) = \text{pow}^L(i)$, $\text{pow}^L(\text{pow}^L(y)) = \text{pow}^L(\text{pow}^L(y)) \cdot \text{pow}^L(\text{pow}^L(y)) = \text{pow}^L(\text{pow}^L(y)) \cdot \text{pow}^L(\text{pow}^L(y)) = \text{pow}^L(\text{pow}^L(y)) = \text{pow}^L(y)$. □

Claim 3.2.62

1. $\epsilon \# y = \epsilon = x \# \epsilon$
2. (L) $1(x \# y) = 1(x \# y) \cdot _1 y$ (R) $1(x \# y) = 1(x \# y) \cdot _1 x$
3. (L) $1((x \cdot y) \# z) = 1(x \# z) \cdot _1 (y \# z)$ (R) $1(x \# (y \cdot z)) = 1(x \# y) \cdot _1 (x \# z)
4. $1(x \# y) = 1(y \# x)$

Proof. Again, all these statements can be proved by very simple applications of NIND or TIND; we give the proof of the third statement (for the left case) as an illustration: $1((x \cdot \epsilon) \# z) = 1(x \# z) = 1(x \# z) \cdot _1 (\epsilon \# z)$, $1((x \cdot y) \# z) = 1((x \cdot y) \# z) \cdot _1 z = 1(x \# z) \cdot _1 (y \# z) \cdot _1 z = 1(x \# z) \cdot _1 (y \# z)$. □

Claim 3.2.63

1. $\text{powdiv}^L(x, y) = \text{powdiv}^L(x, 1 y)$
2. $\text{powdiv}^L(x, y) = \text{powdiv}^L(x, \text{pow}^L(y))$
3. $x \triangleright \text{pow}^L(y) \neq \epsilon \rightarrow \text{powdiv}^L(x, y) = \epsilon$
4. $\text{powdiv}^L(\text{pow}^L(y) \# z, y) = y \triangleright \epsilon \triangleright \epsilon \triangleright \epsilon \triangleright \epsilon$

Proof.

1. By a simple TIND on $y$.

2. By a simple TIND on $y$.

3. First, we prove that $x \triangleright \text{pow}^L(y) \neq \epsilon \rightarrow x \triangleright \text{pow}^L(y) \neq \epsilon$ by proving the contrapositive by TIND on $y$: $x \triangleright \text{pow}^L(x) = \epsilon$ (so the statement is vacuously true), $x \triangleright \text{pow}^L(\epsilon) = \epsilon \rightarrow x \triangleright x \triangleright \epsilon \neq \epsilon \rightarrow x \triangleright \epsilon \triangleright \text{pow}^L(\epsilon) = \epsilon$, and assuming that $x \triangleright \text{pow}^L(\epsilon) = \epsilon$, then

$$x \triangleright y = (x \triangleright \text{pow}^L(\epsilon) \triangleright \text{pow}^L(y))$$

$$= \text{pow}^L(\text{pow}^L(y)) \cdot \text{pow}^L(\text{pow}^L(y))$$

$$= \text{pow}^L(\text{pow}^L(y)) \cdot \text{pow}^L(\text{pow}^L(y))$$
\[= ▶x ▶ ((x ⊲ ▶ pow^L(y)) ▶ pow^L(y))\]
\[= (x ⊲ ▶ pow^L(y)) ▶ (▶x ▶ pow^L(y))\]
\[= (x ⊲ ▶ pow^L(y)) ▶ ▶ \varepsilon = \varepsilon\]

(where we have used the fact that \((z ∙ y) ▶ x = y ▶ (z ▶ x) = z ▶ (y ▶ x)\), which is easy to prove by NIND on \(z\)). The result then follows by a simple application of TIND.

4. By TIND on \(y\): \(\text{powdiv}^L(\text{pow}^L(\varepsilon) \# z, \varepsilon) = \varepsilon = \varepsilon \? 2L (\varepsilon, 1z)\), \(\text{powdiv}^L(\text{pow}^L(i) \# z, i) = \text{powdiv}^L(z, i) = 1z = i \? 2L (\varepsilon, 1z)\), \(\text{powdiv}^L(\text{pow}^L(y) \# z, y) = \text{powdiv}^L((\text{pow}^L(y) \# z) ▶ ▶ y) = \text{powdiv}^L(\text{pow}^L(y) ▶ ▶ z, ▶ y) = \text{powdiv}^L(\text{pow}^L(y) ▶ ▶ z, ▶ y) = 1z = y \? 2L (\varepsilon, 1z)\). □

**Claim 3.2765**

1. \(x ▶ (\text{pow}^L(y) \# \text{powdiv}^L(x, y)) = \varepsilon\)

2. \(y \neq \varepsilon \to \text{powmod}^L(x, y) ▶ \text{pow}^L(y) \neq \varepsilon\)

3. \(y \neq \varepsilon \to \text{powdiv}^L(x, y, 1) = \text{powdiv}^L(x, y) \cdot ((\text{powmod}^L(x, y, 1) ▶ \text{pow}^L(y) \? 2L (1, \varepsilon))\)
\(y \neq \varepsilon \to \text{powmod}^L(x, y, 1) = (\text{powmod}^L(x, y, 1) ▶ \text{pow}^L(y) \? 2L (\varepsilon, \text{powmod}^L(x, y, 1))\)

4. \(\text{powdiv}^L((\text{pow}^L(y) \# z) \cdot x, y) = y \? 2L (\varepsilon, 1z) \cdot \text{powdiv}^L(x, y) \wedge \text{powmod}^L((\text{pow}^L(y) \# z) \cdot x, y) = \text{powmod}^L(x, y)\)

5. \(y \neq \varepsilon \wedge x ▶ \text{pow}^L(y) = \varepsilon \to \text{powdiv}^L(x, y) = \text{powdiv}^L(x ▶ \text{pow}^L(y), y) \cdot 1\)
\(y \neq \varepsilon \wedge x ▶ \text{pow}^L(y) = \varepsilon \to \text{powmod}^L(x, y) = \text{powmod}^L(x ▶ \text{pow}^L(y), y)\)

**Proof**

1. By TIND on \(y\) (with \(h_t = h_r = ▶\)): \(x ▶ (\text{pow}^L(\varepsilon) \# \text{powdiv}^L(x, \varepsilon)) = x ▶ \varepsilon = \varepsilon\), \(x ▶ (\text{pow}^L(i) \# \text{powdiv}^L(x, i)) = x ▶ (1 \# 1x) = \varepsilon\),

\[x ▶ (\text{pow}^L(y) \# \text{powdiv}^L(x, y))\]
\[= x ▶ (\text{pow}^L(y) \# \text{powdiv}^L(x ▶ ▶ y))\]
\[= (x ▶ ▶ ▶ x) ▶ ( (\text{pow}^L(\varepsilon ▶ ▶ y) \cdot \text{pow}^L(\varepsilon ▶ ▶ y)) \# \text{powdiv}^L(x ▶ ▶ y))\]
\[= \varepsilon ▶ (((x ▶ ▶ (\text{pow}^L(\varepsilon ▶ ▶ y) \# \text{powdiv}^L(x ▶ ▶ y))))) \cdot\]
\((x ▶ (\text{pow}^L(\varepsilon ▶ ▶ y) \# \text{powdiv}^L(x ▶ ▶ y)))) ▶ (\text{pow}^L(\varepsilon ▶ ▶ y) \# \text{powdiv}^L(x ▶ ▶ y))\)
\[= (x ▶ (\text{pow}^L(\varepsilon ▶ ▶ y) \# \text{powdiv}^L(x ▶ ▶ y)))) ▶ (\varepsilon ▶ (\text{pow}^L(\varepsilon ▶ ▶ y) \# \text{powdiv}^L(x ▶ ▶ y)))\]
\[= (x ▶ (\text{pow}^L(\varepsilon ▶ ▶ y) \# \text{powdiv}^L(x ▶ ▶ y)))) ▶ \varepsilon = \varepsilon.\]
2. By TIND on \( y \neq \varepsilon \): \( \text{powmod}^L(x, 1) \triangleright \text{pow}^L(1) = \varepsilon \triangleright 1 \neq \varepsilon \),

\[
\text{powmod}^L(x, y) \triangleright \text{pow}^L(y)
\]

\[
= \left( (\text{pow}^L(y) \not\triangleright \text{powdiv}^L(x, y)) \triangleright 1 \right) \triangleright \text{pow}^L(y)
\]

\[
= \left( \left( (\text{pow}^L(\triangleright y) \cdot \text{pow}^L(\triangleright y)) \not\triangleright \text{powdiv}^L(x \triangleright \triangleright y) \right) \triangleright (1 \triangleright \triangleright \triangleright x) \right)
\]

\[
\triangleright (\text{pow}^L(\triangleright y) \cdot \text{pow}^L(\triangleright y))
\]

\[
= \left( \left( (\text{pow}^L(\triangleright y) \not\triangleright \text{powdiv}^L(x \triangleright \triangleright y)) \triangleleft (\text{pow}^L(\triangleright y) \not\triangleright \text{powdiv}^L(x \triangleright \triangleright y)) \right) \triangleleft (1 \triangleright \triangleright \triangleright x \cdot \cdot (x ?^\text{EL} (\varepsilon, 1))) \right) \triangleright (\text{pow}^L(\triangleright y) \cdot \text{pow}^L(\triangleright y))
\]

\[
= \left( \left( (\text{pow}^L(\triangleright y) \not\triangleright \text{powdiv}^L(x \triangleright \triangleright y)) \triangleleft 1 \right) \cdot \cdot (x ?^\text{EL} (\varepsilon, 1))) \right) \triangleright (\text{pow}^L(\triangleright y) \cdot \text{pow}^L(\triangleright y))
\]

\[
= \text{powmod}^L(x \triangleright \triangleright y) \triangleleft \text{pow}^L(\triangleright y) \cdot (\text{powmod}^L(x \triangleright \triangleright y) \cdot (x ?^\text{EL} (\varepsilon, 1))) \triangleright \text{pow}^L(\triangleright y) \neq \varepsilon
\]

(where we have used the fact that \( y < x = \varepsilon \land w < z = \varepsilon \rightarrow wy \triangleright 1z_1x = w \triangleright 1z \cdot y \triangleright 1x \), which is a direct consequence of Claim \([3, 2, 54]\) and the fact that \( zy \triangleright x = y \triangleright (z \triangleright x) \)).

3. We prove the first statement by TIND on \( y \neq \varepsilon \): \( \text{powdiv}^L(x, 1, 1) = x_1 = x \cdot (1 \triangleright 1 ?^\text{ZL} (1, \varepsilon)) = \text{powdiv}^L(x, 1) \cdot ((\text{powmod}^L(x, 1) \cdot 1) \triangleright 1 ?^\text{ZL} (1, \varepsilon)) \), and before proving the inductive case, we can show by TIND on \( y \neq \varepsilon \) that

\[
(\text{powmod}^L(x, y) \cdot 1) \triangleright \text{pow}^L(y) ?^\text{ZL} (1, \varepsilon)
\]

\[
= x ?^\text{EL} \left( \left( (\text{powmod}^L(x \triangleright \triangleright y) \cdot 1) \triangleright \text{pow}^L(\triangleright y) \cdot \text{powmod}^L(x \triangleright \triangleright y) \triangleright \text{pow}^L(\triangleright y) \right) ?^\text{ZL} (1, \varepsilon), \right.
\]

\[
\left. \left( (\text{powmod}^L(x \triangleright \triangleright y) \cdot 1) \triangleright \text{pow}^L(\triangleright y) \cdot (\text{powmod}^L(x \triangleright \triangleright y) \cdot 1) \triangleright \text{pow}^L(\triangleright y) \right) ?^\text{ZL} (1, \varepsilon) \right)
\]

\[
= x ?^\text{EL} \left( \varepsilon, (\text{powmod}^L(x \triangleright \triangleright y) \cdot 1) \triangleright \text{pow}^L(\triangleright y) ?^\text{ZL} (1, \varepsilon) \right)
\]

(the proof is similar to that of the preceding claim), so that

\[
\text{powdiv}^L(x, 1, y)
\]

\[
= \text{powdiv}^L((x1) \triangleright \triangleright y)
\]

\[
= x ?^\text{EL} (\text{powdiv}^L(x \triangleright \triangleright y), \text{powdiv}^L(x \cdot 1, \triangleright y))
\]

\[
= x ?^\text{EL} (\text{powdiv}^L(x, y), \text{powdiv}^L(x \triangleright \triangleright y) \cdot (\text{powmod}^L(x \triangleright \triangleright y) \cdot 1) \triangleright \text{pow}^L(\triangleright y) ?^\text{ZL} (1, \varepsilon))
\]

\[
= \text{powdiv}^L(x, y) \cdot (x ?^\text{EL} (\varepsilon, (\text{powmod}^L(x \triangleright \triangleright y) \cdot 1) \triangleright \text{pow}^L(\triangleright y) ?^\text{ZL} (1, \varepsilon)))
\]

\[
= \text{powdiv}^L(x, y) \cdot ( (\text{powmod}^L(x, y) \cdot 1) \triangleright \text{pow}^L(y) ?^\text{ZL} (1, \varepsilon)).
\]

As for the second statement, it follows directly from the first by the definition of \( \text{powmod}^L \) and the fact that \( \text{powmod}^L(x, y) \triangleright \text{pow}^L(y) \neq \varepsilon \rightarrow \text{pow}^L(y) \triangleright (\text{powmod}^L(x, y) \cdot 1) = \varepsilon \).
4. By NIND on $x$: $\text{powdiv}^L((\text{pow}^L(y) \# z) \cdot \epsilon, y) = y \mho (\epsilon, 1z) = y \mho (\epsilon, 1z) \cdot \text{powdiv}^L(\epsilon, y)$

$\text{powdiv}^L((\text{pow}^L(y) \# z) \cdot x_i, y)$

$= \text{powdiv}^L((\text{pow}^L(y) \# z) \cdot x, y) \cdot (\text{powmod}^L((\text{pow}^L(y) \# z) \cdot x, y) \cdot 1) \supset \text{pow}^L(y) \mho (1, \epsilon)\)

$= (y \mho (\epsilon, 1z)) \cdot \text{powdiv}^L(x, y) \cdot ((\text{powmod}^L(x, y) \cdot 1) \supset \text{pow}^L(y) \mho (1, \epsilon))$

$= (y \mho (\epsilon, 1z)) \cdot \text{powdiv}^L(x_1, y)$

$\text{powmod}^L((\text{pow}^L(y) \# z) \cdot x_i, y)$

$= (\text{pow}^L(y) \# \text{powdiv}^L((\text{pow}^L(y) \# z) \cdot x_i, y)) \supset 1((\text{pow}^L(y) \# z) \cdot x_i)$

$= (\text{pow}^L(y) \# ((y \mho (\epsilon, 1z)) \cdot \text{powdiv}^L(x_i, y))) \supset 1(\text{pow}^L(y) \# z) \cdot 1(x_i)$

$= (\text{pow}^L(y) \# (y \mho (\epsilon, 1z))) \cdot (\text{pow}^L(y) \# \text{powdiv}^L(x_i, y)) \supset (\text{pow}^L(y) \# 1z) \cdot 1(x_i)$

$= \text{pow}^L(y) \# \text{powdiv}^L(x_i, y) \supset 1(x_i)$

$= \text{powmod}^L(x_i, y)$.

5. By NIND on $x$: $y \neq \epsilon \land \epsilon \supset \text{pow}^L(y) = \epsilon$ is false so the statement is vacuously true, and $y \neq \epsilon \land x \supset \text{pow}^L(y) = \epsilon \rightarrow (y \neq \epsilon \land x \supset \text{pow}^L(y) = \epsilon) \lor (y \neq \epsilon \land x \supset \text{pow}^L(y) = 1)$, so we prove the statement by cases.

First, if $y \neq \epsilon \land x \supset \text{pow}^L(y) = \epsilon$ (which implies that $1(x_1 \lhd \text{pow}^L(y)) = 1(x \lhd \text{pow}^L(y)) \cdot 1$),

$\text{powdiv}^L(x_1, y)$

$= \text{powdiv}^L(x, y) \cdot ((\text{powmod}^L(x, y) \cdot 1) \supset \text{pow}^L(y) \mho (1, \epsilon))$

$= \text{powdiv}^L(x \lhd \text{pow}^L(y), y) \cdot 1 \cdot ((\text{powmod}^L(x \lhd \text{pow}^L(y)) \cdot 1) \supset \text{pow}^L(y) \mho (1, \epsilon))$

$= \text{powdiv}^L((x \lhd \text{pow}^L(y)) \cdot 1, y) \cdot 1 = \text{powdiv}^L(x_1 \lhd \text{pow}^L(y), y) \cdot 1,$

$\text{powmod}^L(x_1, y)$

$= (\text{powmod}^L(x, y) \cdot 1) \supset \text{pow}^L(y) \mho (\epsilon, \text{powmod}^L(x, y) \cdot 1)$

$= (\text{powmod}^L(x \lhd \text{pow}^L(y), y) \cdot 1) \supset \text{pow}^L(y) \mho (\epsilon, \text{powmod}^L(x \lhd \text{pow}^L(y), y) \cdot 1)$

$= \text{powmod}^L((x \lhd \text{pow}^L(y)) \cdot 1, y) = \text{powmod}^L(x_1 \lhd \text{pow}^L(y), y)$.

Second, if $y \neq \epsilon \land x \supset \text{pow}^L(y) = 1$ (which implies that $1(x_1 \lhd \text{pow}^L(y)) = \epsilon$),

$\text{powdiv}^L(x_1, y) = \text{powdiv}^L(x, y) \cdot ((\text{powmod}^L(x, y) \cdot 1) \supset \text{pow}^L(y) \mho (1, \epsilon))$

$= \epsilon \cdot ((x_1) \supset \text{pow}^L(y) \mho (1, \epsilon))$

$= 1 = \text{powdiv}^L(\epsilon, y) \cdot 1$

$= \text{powdiv}^L(x_1 \lhd \text{pow}^L(y), y) \cdot 1$,
\( \text{powmod}^L(x_1, y) = (\text{powmod}^L(x, y) \cdot 1) \triangleright \text{pow}^L(y)^{2L}(\varepsilon, \text{powmod}^L(x, y) \cdot 1) \)
\[ = x_1 \triangleright \text{pow}^L(y)^{2L}(\varepsilon, 1x_1) \]
\[ = \varepsilon = \text{powmod}^L(\varepsilon, y) \]
\[ = \text{powmod}^L(x_1 \triangleright \text{pow}^L(y), y). \text{ } \square \]

On "\(=^N\)" and "\(<^N\)"

**Theorem 3.2.13**

1. \(x =^N \varepsilon \iff x = 0x\)
2. \(x =^N y \iff \text{l}_{p_0}(x, y) = \text{l}_{p_0}(y, x)\)

**Proof**

1. By a straightforward NIND on \(x\).
2. By the preceding property and Derived Rule 3.2.3: \(x =^N \varepsilon \iff x = 0x \iff \text{l}_{p_0}(x, \varepsilon) = \text{l}_{p_0}(\varepsilon, x)\)
   (and similarly for \(\varepsilon =^N y\), \(x =^N y \iff x =^N y \land^B (i \iff^B j) \iff \text{l}_{p_0}(x, y) = \text{l}_{p_0}(y, x) \land i = j \iff \text{l}_{p_0}(x, y) \cdot i = \text{l}_{p_0}(y, x) \cdot j \iff \text{l}_{p_0}(x_i, y_j) = \text{l}_{p_0}(y_j, x_i)\). \text{ } \square

**Claim 3.2.67**

1. \(x_0 =^N y_0 \iff x =^N y \iff x_1 =^N y_1\)
2. \(\neg^B(x_0 =^N y_1)\)
3. \(\neg^B(x_1 =^N y_0)\)

**Proof** Directly from Theorem 3.2.13: \(x_1 =^N y_j \iff \text{l}_{p_0}(x_i, y_j) = \text{l}_{p_0}(y_j, x_i) \iff \text{l}_{p_0}(x, y) \cdot i = \text{l}_{p_0}(y, x) \cdot j \iff \text{l}_{p_0}(x_i, y_j) = \text{l}_{p_0}(y_j, x_i) \land i = j \iff x =^N y \land i = j. \text{ } \square

**Claim 3.2.68**

1. \(x_0 <^N y_0 = x <^N y \lor^B (x =^N y \land^B 0 <^B 0) = x <^N y\)
2. \(x_0 <^N y_1 = x <^N y \lor^B (x =^N y \land^B 0 <^B 1) = x <^N y\)
3. \(x_1 <^N y_0 = x <^N y \lor^B (x =^N y \land^B 1 <^B 0) = x <^N y\)
4. \(x_1 <^N y_1 = x <^N y \lor^B (x =^N y \land^B 1 <^B 1) = x <^N y\)

**Proof** From Theorem 3.2.10 we have that \(x_i <^N y_j = x <^N y \lor^B (x =^N y \land^B i <^B j)\), so the theorem follows directly from the preceding claim. \text{ } \square
ON $\equiv^N$ AND $\prec^N$

**Claim 3.2.69**

1. $\neg B(x <^N \varepsilon)$
2. $\neg B(x <^N x)$
3. $\neg B(\varepsilon <^N 0x)$

**Proof** A straightforward proof by NIND, using the preceding claims. □

**Lemma 3.2.72**

1. $x =^N y = 0x =^N y = x =^N 0y = 0x =^N 0y$
2. $x <^N y = 0x <^N 0y$
3. $x <^N y = 0x <^N y = x <^N 0y$

**Proof**

1. Direct from the fact that $(\theta_0(0x, y) = \theta_0(x, y)) \lor (\theta_0(0x, y) = 0 \cdot \theta_0(x, y))$ (which can easily be proved by cases depending on the length of $x \succ y$), and by Derived Rule 3.2.3.

2. By a simple application of Derived Rule 3.2.3 and the first claim (we show only the inductive step, the base cases being just as simple): $0xi <^N 0yj = 0x <^N 0y \lor^B (0x =^N 0y \land^B i <^B j) = x <^N y \lor^B (x =^N y \land^B i <^B j) = xi <^N yj$.

3. By the second claim:

$$0x <^N y = 0x \prec y \Rightarrow (0(y \prec 0x) \cdot 0x <^N y) \lor (0x <^N 0(y \prec x) \cdot y) = x \succ y \Rightarrow (0(x <^N 0 \cdot 0(x \prec y) \cdot y) = x \succ y \Rightarrow (x <^N 0(x \prec y) \cdot y) = x <^N y$$

(and similarly for $x <^N 0y$). □

**Theorem 3.2.15** $x =^N y \land y <^N z \rightarrow x <^N z$ and $x =^N y \land y >^N z \rightarrow x >^N z$

**Proof** Follows directly from the (already proven) facts that $x =^N y \leftrightarrow \theta_0(x, \max_3^1(x, y, z)) = \theta_0(y, \max_3^1(x, y, z))$ and $y <^N z \leftrightarrow \theta_0(y, \max_3^1(x, y, z)) <^N \theta_0(z, \max_3^1(x, y, z))$ □

**Corollary 3.2.73** $x =^N y \land y \leq^N z \rightarrow x \leq^N z$ and $x =^N y \land y \geq^N z \rightarrow x \geq^N z$

**Proof** Directly from the theorem. □
Theorem 3.2.10 \( x ^N y \land y ^N z \rightarrow x ^N z \quad \text{and} \quad x ^N y \land y ^N z \rightarrow x ^N z \)

Proof By Derived Rule 3.2.23, the base cases for \( z = \varepsilon \) and \( y = \varepsilon \) are trivially true since \( \varepsilon \not< x \), while the third base case \( \varepsilon \not< y \land y \not< z \rightarrow \varepsilon \not< z \) is itself proved by Derived Rule 3.2.23. The two base cases are again trivially true, and \( \varepsilon \not< y \land \varepsilon \not< j \land \varepsilon \not< k = (\varepsilon \not< y \lor \varepsilon \not< j \lor \varepsilon \not< k) \lor (\varepsilon \not< y \land \varepsilon \not< j \land \varepsilon \not< k) \lor (\varepsilon \not< y \land \varepsilon \not< j \land \varepsilon \not< k) = \varepsilon \not< \varepsilon \lor \varepsilon \not< j \lor \varepsilon \not< k \lor \varepsilon \not< \varepsilon \lor \varepsilon \not< j \lor \varepsilon \not< k \lor \varepsilon \not< \varepsilon \lor \varepsilon \not< j \lor \varepsilon \not< k \
\) (the general inductive step is almost identical). \( \Box \)

Corollary 3.2.73 \( x ^N y \land y ^N z \rightarrow x ^N z \quad \text{and} \quad x ^N y \land y ^N z \rightarrow x ^N z \)

Proof Directly from the theorem. \( \Box \)

Corollary 3.2.75 \( x ^N y \land y ^N z \rightarrow x ^N z \quad \text{and} \quad x ^N y \land y ^N z \rightarrow x ^N z \)

Proof Directly from the theorem. \( \Box \)

On “\(|\cdot|\)” and “\(\text{succ}^N\)”

Claim 3.2.76

\[
\text{succ}^N(\varepsilon) = 1 \\
\text{succ}^N(x0) = 0x1 \\
\text{succ}^N(x1) = \text{succ}^N(x) \cdot 0
\]

Proof Simple proofs by NIND (proving the relevant properties first for the auxiliary function \(\text{cuss}^N\), and then for \(\text{succ}^N\)). \( \Box \)

Claim 3.2.77

\[
\text{succ}^N(0x) = 0 \cdot \text{succ}^N(x) \\
\text{succ}^N(1x) = '\text{succ}^N(x) \cdot \neg^B\text{succ}^N(x) \cdot \text{succ}^N(x) \\
'\text{succ}^N(x) = \text{AND}(1x) \\
\text{succ}^N(x) = \text{AND}(1x) \ ?^B (1 \cdot 0x, 0 \cdot \text{succ}^N(x))
\]

Proof All the proofs are simple, but we will illustrate them by proving the second property, by NIND on \( x \): \(\text{succ}^N(1\varepsilon) = 10 = 1 \cdot \neg^B1 \cdot \varepsilon = '\text{succ}^N(\varepsilon) \cdot \neg^B\text{succ}^N(\varepsilon) \cdot \text{succ}^N(\varepsilon) = \text{succ}^N(1x0) = 01x1 = '0x1) \cdot \neg^B(0x1) \cdot \varepsilon = '\text{succ}^N(x0) \cdot \neg^B\text{succ}^N(x0) \cdot \text{succ}^N(x0) = \text{succ}^N(1x1) = \text{succ}^N(1x) \cdot 0 = '\text{succ}^N(x) \cdot \neg^B\text{succ}^N(x) \cdot \text{succ}^N(x) \cdot 0 = '\text{succ}^N(x) \cdot 0 \cdot \neg^B(\text{succ}^N(x) \cdot 0) \cdot \text{succ}^N(x) \cdot 0 = '\text{succ}^N(x1) \cdot \neg^B\text{succ}^N(x1) \cdot \text{succ}^N(x1). \ \Box
Theorem 3.2.17

1. \(x <^N \text{succ}^N(x)\)

2. \(x >^N y = x \geq^N \text{succ}^N(y)\)

Proof

1. A simple direct proof by NIND: \(\varepsilon <^N \text{succ}^N(\varepsilon) = \varepsilon <^N 1\), \(x0 <^N \text{succ}^N(0) = 0 <^N 0x1 = x \leq^N 0x\), \(x1 <^N \text{succ}^N(1) = x1 <^N \text{succ}^N(x) \cdot 0 = x <^N \text{succ}^N(x)\).

2. By Derived Rule 3.2.3: \(\varepsilon >^N y = 0 = \varepsilon \geq^N \text{succ}^N(y)\) (since \(\text{succ}^N(y) \neq^N \varepsilon\)), \(x >^N \varepsilon = x \geq^N \text{succ}^N(\varepsilon)\) (proved by an easy NIND on \(x\)),

\[
\begin{align*}
  xi >^N y0 &= x >^N y \lor^B (x =^N y \land^B i >^B 0) \\
  &= x >^N y \lor^B (x =^N y \land^B i) \\
  &= x >^N 0y \lor^B (x =^N 0y \land^B i >^B 1) \lor^B (x =^N 0y \land^B i) \\
  &= xi >^N 0y \lor^B xi =^N 0y \\
  &= xi >^N 0y = xi \geq^N \text{succ}^N(y0),
\end{align*}
\]

\[
\begin{align*}
  xi >^N y1 &= x >^N y \lor^B (x =^N y \land^B i >^B 1) \\
  &= x >^N y \\
  &= x \geq^N \text{succ}^N(y) \\
  &= x >^N \text{succ}^N(y) \lor^B (x =^N y \land^B i >^B 0) \\
  &= x >^N \text{succ}^N(y) \lor^B (x =^N y \land^B i >^B 0) \lor^B (x =^N y \land^B i =^B 0) \\
  &= xi >^N \text{succ}^N(y) \lor^B xi =^N \text{succ}^N(y) \cdot 0 \\
  &= xi >^N \text{succ}^N(y) \cdot 0 = xi >^N \text{succ}^N(y1).\quad \square
\end{align*}
\]

Claim 3.2.78 \(|x| = |jx|\)

Proof By TIND on \(x\): \(|\varepsilon| = \varepsilon = |j| = |j| = |1| = |1| = |x| = x =^E (|x| \cdot 0, |x| \cdot 1) = jx \cdot E (j, 0, 1) = jx| = |jx|\). \(\square\)

Theorem 3.2.19 \(|xi| =^N \text{succ}^N(|x|)\)
Proof. By TIND on $x$: $|\varepsilon i| = 1 = \text{succ}^N(|\varepsilon|)$, $|ji| = 10 = \text{succ}^N(1) = \text{succ}^N(|j|)$,

$$
|x| = x ?^E (|(xi)| \cdot 0, (|xi|) \cdot 1)
= x ?^E (|(xi)| \cdot 1, (|xi|) \cdot 0)
= x ?^E (|x| \cdot 1, |x| \cdot j \cdot 0)
= N \cdot x ?^E (\text{succ}^N(|x|) \cdot 0, \text{succ}^N(|x|) \cdot 0)
= x ?^E (\text{succ}^N(|x|) \cdot 0, \text{succ}^N(|x|) \cdot 1)
= \text{succ}^N(|x|)
$$

(where the fifth equality, where "\(=_{N}\)" is introduced, holds by the induction hypothesis). □

On “masking” functions

Theorem 3.2.20

$$
\begin{align*}
\text{first}_0(0x) &= 1 \cdot 0x \\
\text{first}_0(1x) &= 0 \cdot \text{first}_0(x) \\
\text{first}_1(0x) &= 0 \cdot \text{first}_1(x) \\
\text{first}_1(1x) &= 1 \cdot 0x
\end{align*}
$$

Proof. By a straightforward NIND. □

Corollary 3.2.81

$$
\text{first}_0(x) = \text{first}_1(\text{not}^B(x))
$$

Proof. By a straightforward NIND, from the preceding theorem. □

On binary addition

Theorem 3.2.21

$$
x +^N y = y +^N x
$$

Proof. Direct from the commutativity of the functions involved (i.e., \(\text{xor}^B\) and \(\text{and}^B\)). □

Lemma 3.2.82

$$
\begin{align*}
carry^N(x0, 1) &= carry^N(x, \varepsilon) \cdot 0 = a0x \\
carry^N(x1, 1) &= x ?^E (1, carry^N(x, 1) \cdot 1)
\end{align*}
$$
On binary addition

Proof (We prove only the second statement, the first is a simple application of NIND on \(x\).) By NIND on \(x\): \(\text{carry}^N(\varepsilon \cdot 1, 1) = 1 = \varepsilon \oslash 2^L (1, \text{carry}^N(\varepsilon, 1) \cdot 1),\)

\[
\text{carry}^N(0x1, 1) = 0 \cdot \text{carry}^N(x1, 1) \\
= 0 \cdot x \oslash 2^L (1, \text{carry}^N(x, 1) \cdot 1) \\
= x \oslash 2^L (0 \cdot 1, 0 \cdot \text{carry}^N(x, 1) \cdot 1) \\
= x \oslash 2^L (\text{carry}^N(0, 1) \cdot 1, \text{carry}^N(0x, 1) \cdot 1) \\
= \text{carry}^N(0x, 1) \cdot 1,
\]

\[
\text{carry}^N(1x1, 1) = \text{carry}^N(x1, 1) \cdot \text{carry}^N(x1, 1) \\
= x \oslash 2^L (\varepsilon^2 1, \text{carry}^N(x, 1) \cdot 1) \cdot \text{carry}^N(x, 1) \cdot 1) \\
= x \oslash 2^L (1, \text{carry}^N(x, 1) \cdot \text{carry}^N(x, 1) \cdot 1) \\
= x \oslash 2^L (\text{carry}^N(1, 1) \cdot 1, \text{carry}^N(1x, 1) \cdot 1) \\
= \text{carry}^N(1x, 1) \cdot 1.
\]

\[\square\]

Theorem 3.2.22 \(x + 1 = x \oslash 2^L (0 \cdot \text{succ}^N(x), \text{succ}^N(x)) = \text{succ}^N(x)\)

Proof By NIND on \(x\): \(\varepsilon + 1 = 01 = \varepsilon \oslash 2^L (0 \cdot \text{succ}^N(\varepsilon), \text{succ}^N(\varepsilon)),\)

\[
x0 + 1 = \text{xor}^B_3(\text{carry}^N(x0, 1) \cdot 0, x0, 1) \\
= \text{xor}^B_3(0x0, 0x0, 0x0, 0x1) \\
= 0x1 \\
= \text{succ}^N(x0),
\]

\[
x1 + 1 = \text{xor}^B_3(\text{carry}^N(x1, 1) \cdot 0, x1, 1) \\
= \text{xor}^B_3(\text{carry}^N(x, 1) \cdot 0, x1, 0x1) \\
= \text{xor}^B_3(\text{carry}^N(x, 1) \cdot 1, 0x1, 0x0) \cdot 0 \\
= \text{xor}^B_3(\text{carry}^N(x, 1) \cdot 0, 0x1, 0x0) \cdot 0 \\
= \text{xor}^B_3(\text{carry}^N(x, 1) \cdot 0, x, 1) \cdot 0 \\
= (x + 1) \cdot 0 \\
= \text{succ}^N(x) \cdot 0 \\
= \text{succ}^N(x1). \quad \square
\]
Claim 3.2.83  

For \(j x = j y\),

\[
\begin{align*}
\text{carry}^N(x, \varepsilon) &= 0 \cdot 0 x \\
\text{carry}^N(x, x) &= x0 \\
\text{carry}^N(0x, 0y) &= 0 \cdot \text{carry}^N(x, y) \\
\text{carry}^N(1x, 0y) &= \text{carry}^N(x, y) \cdot \text{carry}^N(x, y) \\
\text{carry}^N(1x, 1y) &= 1 \cdot \text{carry}^N(x, y)
\end{align*}
\]

Proof  

All these properties can be proved with a simple application of Derived Rule 3.2.6, or directly from the definition of \(\text{carry}^N\). The last three depend on the following facts.

\[
\begin{align*}
\text{maskbit}(\text{and}_2^{0}(0x, 0y), \text{first}_0(\text{xor}_2^{0}(0x, 0y))) &= \text{maskbit}(0 \cdot \text{and}_2^{0}(x, y), \text{first}_0(0 \cdot \text{xor}_2^{0}(x, y))) \\
&= \text{maskbit}(0 \cdot \text{and}_2^{0}(x, y), 1 \cdot 0 \text{xor}_2^{0}(x, y)) \\
&= 0 \\
\text{maskbit}(\text{and}_2^{0}(1x, 0y), \text{first}_0(\text{xor}_2^{0}(1x, 0y))) &= \text{maskbit}(0 \cdot \text{and}_2^{0}(x, y), \text{first}_0(1 \cdot \text{xor}_2^{0}(x, y))) \\
&= \text{maskbit}(0 \cdot \text{and}_2^{0}(x, y), 0 \cdot \text{first}_0(\text{xor}_2^{0}(x, y))) \\
&= \text{maskbit}(\text{and}_2^{0}(x, y), \text{first}_0(\text{xor}_2^{0}(x, y))) \\
\text{maskbit}(\text{and}_2^{0}(1x, 1y), \text{first}_0(\text{xor}_2^{0}(1x, 1y))) &= \text{maskbit}(1 \cdot \text{and}_2^{0}(x, y), \text{first}_0(0 \cdot \text{xor}_2^{0}(x, y))) \\
&= \text{maskbit}(1 \cdot \text{and}_2^{0}(x, y), 1 \cdot 0 \text{xor}_2^{0}(x, y)) \\
&= 1 \quad \Box
\end{align*}
\]

Claim 3.2.84  

For \(j x = j y\),

\[
\begin{align*}
\text{carry}^N(x, \varepsilon) &= 0 \cdot 0 x \\
\text{carry}^N(x, x) &= x0 \\
\text{carry}^N(0x, 0y) &= 0 \cdot \text{carry}^N(x, y) \\
\text{carry}^N(1x, 0y) &= \text{carry}^N(x, y) \cdot \text{carry}^N(x, y) \\
\text{carry}^N(1x, 1y) &= 1 \cdot \text{carry}^N(x, y)
\end{align*}
\]

Proof  

Directly from the corresponding properties for \(\text{carry}^N\), where we have used the fact that \(\text{carry}^N(x, y) = \text{carry}^N(x, y)\). Note that the second property implies that \(\text{succ}^N(x + x) = x1\). \(\Box\)
Claim 3.2.84

\[ \begin{align*}
x0 + N y0 &= (x + N y) \cdot 0 \\
x1 + N y0 &= (x + N y) \cdot 1 \\
x1 + N y1 &= \succ^N(x + N y) \cdot 0
\end{align*} \]

Proof. The first two properties follow directly from the Claim above by Derived Rule 3.2.6.
We prove the third property because it is more involved. First, note that \( \neg \mu \text{AND}(x + N y) \) can be proved directly from Claim 3.2.84 by Derived Rule 3.2.6. This implies that \( \succ^N(x + N y) = 0 \),
which in turn implies that \( \succ^N(1 \cdot (x + N y)) = 01 \cdot \succ^N(x + N y). \) Now, we prove the third property by Derived Rule 3.2.6

\[ 1 + N 1 = 10 = \succ^N(0) \cdot 0 = \succ^N(0) \cdot 0 = \succ^N(x + N \varepsilon) \cdot 0, \]

\[ 0x1 + N 0y1 \]
\[ = 0 \cdot (x1 + N y1) \]
\[ = 0 \cdot \succ^N(x + N y) \cdot 0 \]
\[ = \succ^N(x + N y) \cdot 0 \]
\[ = \succ(0 \cdot \succ^N(x + N y)) \cdot 0 \]
\[ = \succ^N(0 \cdot (x + N y)) \cdot 0 \]
\[ = \succ^N(0x + N 0y) \cdot 0, \]

\[ 1x1 + N 1y1 \]
\[ = 1 \cdot (x1 + N y1) \]
\[ = 1 \cdot \succ^N(x + N y) \cdot 0 \]
\[ = \succ^N(1 \cdot (x + N y)) \cdot 0 \]
\[ = \succ^N(1 \cdot (x + N y)) \cdot 0 \]
\[ = \succ^N(1x + N 1y) \cdot 0, \]

\[ 1x1 + N 0y1 \]
\[ = \succ^N(x + N y) \cdot \neg \mu (x1 + N y1) \cdot \succ^N(x1 + N y1) \]
\[ = \succ^N(x + N y) \cdot \neg \mu (\succ^N(x + N y) \cdot 0) \cdot \succ^N(x + N y) \cdot 0 \]
\[ = \succ^N(x + N y) \cdot \succ^N(x + N y) \cdot \succ^N(x + N y) \cdot 0 \]
\[ = \succ^N(x + N y) \cdot \succ^N(x + N y) \cdot \succ^N(x + N y) \cdot 0 \]
\[ = \succ^N(x + N y) \cdot \succ^N(x + N y) \cdot \succ^N(x + N y) \cdot 0 \]
\[ = \succ^N(x + N y) \cdot \succ^N(x + N y) \cdot \succ^N(x + N y) \cdot 0 \]
\[ = \succ^N(x + N y) \cdot \succ^N(x + N y) \cdot \succ^N(x + N y) \cdot 0 \]
\[ = \succ^N(x + N y) \cdot \succ^N(x + N y) \cdot \succ^N(x + N y) \cdot 0 \]
Theorem 3.2.23

1. $x + ^N y = ^N succ^N(x + ^N y)$
2. $x + ^N (y + ^N z) = ^N (x + ^N y) + ^N z$
3. $y < ^N z \iff x + ^N y < ^N x + ^N z$
4. $x =^N y \land z =^N w \to x + ^N z =^N y + ^N w$
5. $x =^N y \land z <^N w \to x + ^N z <^N y + ^N w$
6. $x <^N y \land z <^N w \to x + ^N z <^N y + ^N w$

Proof

1. By Derived Rule 3.2.23 $\varepsilon + ^N succ^N(y) = ^N succ^N(y) = ^N succ^N(\varepsilon + ^N y)$, $x + ^N succ^N(\varepsilon) = x + ^N 1 = ^N succ^N(x) = ^N succ^N(x + ^N \varepsilon)$,

   \[
   x0 + ^N succ^N(y0) = ^N x0 + ^N y1 \\
   = ^N (x + ^N y) \cdot 1 \\
   = ^N succ^N((x + ^N y) \cdot 0) \\
   = ^N succ^N(x0 + ^N y0),
   \]

   \[
   x0 + ^N succ^N(y1) = ^N x0 + ^N succ^N(y) \cdot 0 \\
   = ^N (x + ^N succ^N(y)) \cdot 0 \\
   = ^N succ^N(x + ^N y) \cdot 0 \\
   = ^N succ^N((x + ^N y) \cdot 1) \\
   = ^N succ^N(x0 + ^N y1),
   \]

   \[
   x1 + ^N succ^N(y0) = ^N x1 + ^N y1 \\
   = ^N succ^N(x + ^N y) \cdot 0 \\
   = ^N succ^N((x + ^N y) \cdot 1) \\
   = ^N succ^N(x1 + ^N y0),
   \]
\[ x_1 + ^N \text{succ}^N(y_1) = ^N x_1 + ^N \text{succ}^N(y) \cdot 0 \]
\[ = ^N (x + ^N \text{succ}^N(y)) \cdot 1 \]
\[ = ^N \text{succ}^N(x + ^N y) \cdot 1 \]
\[ = ^N \text{succ}^N(\text{succ}^N(x + ^N y) \cdot 0) \]
\[ = ^N \text{succ}^N(x_1 + ^N y_1). \]

2. By Derived Rule 3.2.3 generalized to three variables: \( \varepsilon + ^N (y + ^N z) = ^N y + ^N z = ^N (\varepsilon + ^N y) + ^N z \)
(and similarly for \( x, \varepsilon, z \) and \( x, y, \varepsilon \)), \( x_0 + ^N (y_0 + ^N z_0) = ^N x_0 + ^N (y + ^N z) = ^N (x + ^N (y + ^N z)) 0 = ^N ((x + ^N y) + ^N z) 0 = ^N (x + ^N y) 0 + ^N z 0 = ^N (x + ^N y) 0 + ^N z 0, x_1 + ^N (y_0 + ^N z_0) = ^N x_1 + ^N (y + ^N z) 0 = ^N (x + ^N y) 0 + ^N z 0 = ^N (x_1 + ^N y_0) + ^N z 0 \)
(and similarly for \( x_0, y_1, z_0 \) and \( x_1, y_0, z_1 \)), \( x_0 + ^N (y_1 + ^N z_1) = x_0 + ^N \text{succ}^N(y + ^N z) 0 = ^N (x + ^N y) 0 + ^N z 1 \)
(and similarly for \( x_0, y_0, z_1 \) and \( x_1, y_1, z_0 \)), \( x_1 + ^N (y_1 + ^N z_1) = ^N x_1 + ^N \text{succ}^N(y + ^N z) 0 = ^N (x + ^N y) 0 + ^N z 1 \)

3–6. Similar to the last case, straightforward, case-by-case proofs by Derived Rule 3.2.3.

**On iterated sums**

**Claim 3.2.86**

\[ C\text{Scar}_3(x_0, y_0, z_0) = C\text{Scar}_3(x, y, z) \cdot 0 \]
\[ C\text{Scar}_3(x, y_0, z_0) = C\text{Scar}_3(x_1, y_0, z_0) = C\text{Scar}_3(x_1, y_0, z_1) = C\text{Scar}_3(x, y, z) \cdot 0 \]
\[ C\text{Scar}_3(x_0, y_1, z_1) = C\text{Scar}_3(x_1, y_1, z_1) = C\text{Scar}_3(x, y, z) \cdot 1 \]
\[ C\text{Scar}_3(x_1, y_1, z_1) = C\text{Scar}_3(x, y, z) \cdot 1 \]
\[ 0 \cdot C\text{Scar}_3(x, y, z) \cdot 0 = 0 \cdot C\text{Sadd}_3(x, y, z) = 0 \cdot 0 \text{max}_3^L(x, y, z) = 0 \cdot 0 \text{max}_3^L(x_0, y_0, 0, z) \]
\[ 0 \cdot \text{max}_3^L(x_0, y_0, 0, z) = C\text{Scar}_3(x_0, y_0, 0, z) \cdot 0 = C\text{Sadd}_3(x_0, y_0, 0, z) \]
\[ 0 \cdot C\text{Sadd}(x, y, z, w) = 0 \cdot C\text{Sadd}(x, y, z, w) = 0 \cdot 0 \text{max}_4^L(x, y, z, w) = 0 \cdot 0 \text{max}_4^L(x_0, y_0, 0, z, 0, w) \]
\[ 0 \cdot \text{max}_4^L(x_0, y_0, 0, z, 0, w) = C\text{Sadd}(x_0, y_0, 0, z, 0, w) \]
\[ = C\text{Sadd}_3(x_0, y_0, 0, z, 0, w) \]

**Proof** All can be proved with a very simple application of Derived Rule 3.2.6 generalized to three variables, directly from the definitions of the functions involved.

**Lemma 3.2.87**

\[ (C\text{Scar}_3(\text{succ}^N(x), y, z) \cdot 0) + ^N C\text{Sadd}_3(\text{succ}^N(x), y, z) \]
\[ = ^N \text{succ}^N(0) \cdot (C\text{Scar}_3(x, y, z) \cdot 0) + ^N C\text{Sadd}_3(x, y, z) \]
Proof The proof is a straightforward, if tedious, application of Derived Rule 3.2.3 (generalized to three variables). First, the base cases for $\varepsilon, y, z$ and $x, \varepsilon, z$ and $x, y, \varepsilon$ are proved (each one with another application of Derived Rule 3.2.3), and then the eight cases from $x_0, y_0, z_0$ to $x_1, y_1, z_1$ are proved from the assumption that the lemma holds for $x, y, z$. We do not show the full proof here as it is not particularly instructive; instead, we give parts of the proof for two illustrative cases. First, in the proof of the base case ($\text{CScar}_3(\text{succ}^N(\varepsilon, y, z) \cdot 0) + ^N \text{CSadd}_3(\text{succ}^N(\varepsilon), y, z) = ^N \text{succ}^N((\text{CScar}_3(\varepsilon, y, z) \cdot 0) + ^N \text{CSadd}_3(\varepsilon, y, z))$ by Derived Rule 3.2.3, we show the case for $y_1, z_1$.

\[
(\text{CScar}_3(\text{succ}^N(\varepsilon), y_1, z_1) \cdot 0) + ^N \text{CSadd}_3(\text{succ}^N(\varepsilon), y_1, z_1)
\]

\[
= ^N (\text{CScar}_3(1, y_1, z_1) \cdot 0) + ^N \text{CSadd}_3(1, y_1, z_1)
\]

\[
= ^N (\text{CScar}_3(\varepsilon, y, z) \cdot 10) + ^N (\text{CSadd}_3(\varepsilon, y, z) \cdot 1)
\]

\[
= ^N ((\text{CScar}_3(\varepsilon, y, z) \cdot 1) + ^N \text{CSadd}_3(\varepsilon, y, z)) \cdot 1
\]

\[
= ^N \text{succ}^N(((\text{CScar}_3(\varepsilon, y, z) \cdot 1) + ^N \text{CSadd}_3(\varepsilon, y, z)) \cdot 0)
\]

\[
= ^N \text{succ}^N((\text{CScar}_3(\varepsilon, y_1, z_1) \cdot 0) + ^N \text{CSadd}_3(\varepsilon, y_1, z_1))
\]

Second, in the inductive step, we show the case for $x_0, y_1, z_0$.

\[
(\text{CScar}_3(\text{succ}^N(x_0), y_1, z_0) \cdot 0) + ^N \text{CSadd}_3(\text{succ}^N(x_0), y_1, z_0)
\]

\[
= ^N (\text{CScar}_3(x_1, y_1, z_0) \cdot 0) + ^N \text{CSadd}_3(x_1, y_1, z_0)
\]

\[
= ^N (\text{CScar}_3(x, y, z) \cdot 10) + ^N (\text{CSadd}_3(x, y, z) \cdot 0)
\]

\[
= ^N (\text{succ}^N((\text{CScar}_3(x, y, z) \cdot 0) + ^N \text{CSadd}_3(x, y, z)) \cdot 0)
\]

\[
= ^N \text{succ}^N(((\text{CScar}_3(x, y, z) \cdot 0) + ^N \text{CSadd}_3(x, y, z)) \cdot 1)
\]

\[
= ^N \text{succ}^N((\text{CScar}_3(x, y_1, z_0) \cdot 0) + ^N \text{CSadd}_3(x, y_1, z_0))
\]

\[
= ^N \text{succ}^N((\text{CScar}_3(x_0, y_1, z_0) \cdot 0) + ^N \text{CSadd}_3(x_0, y_1, z_0)) \quad \Box
\]

Theorem 3.2.24 $\text{CScar}(x, y, z, w) + ^N \text{CSadd}(x, y, z, w) = ^N x + ^N y + ^N z + ^N w$

Proof As for the preceding lemma, the proof is a straightforward, if tedious, application of Derived Rule 3.2.3 (generalized to four variables). First, the base cases for $\varepsilon, y, z, w$ and $x, \varepsilon, w$ and $x, y, \varepsilon, w$ and $x, y, z, \varepsilon$ are proved (each one with another application of Derived Rule 3.2.3), and then the sixteen cases from $x_0, y_0, z_0, w_0$ to $x_1, y_1, z_1, w_1$ are proved from the assumption that the lemma holds for $x, y, z, w$. We do not show the full proof here as it is not particularly instructive; instead, we give parts of the proof for one illustrative case. In the
inductive step, we show the case for $x_1, y_1, z_1, w_1$.

\[
\begin{align*}
\text{CScar}(x_1, y_1, z_1, w_1) +^N \text{CSadd}(x_1, y_1, z_1, w_1) \\
=^N \text{CScar}_3(\text{CScar}_3(x, y, z) \cdot 10, \text{CSadd}_3(x, y, z) \cdot 1, w_1) \cdot 0 +^N \text{CSadd}_3(\text{CScar}_3(x, y, z) \cdot 10, \text{CSadd}_3(x, y, z) \cdot 1, w_1) \\
=^N \text{CScar}_3(\text{succ}^N(\text{CScar}(x, y, z) \cdot 0), \text{CSadd}_3(x, y, z), w) \cdot 0 +^N \text{CSadd}_3(\text{succ}^N(\text{CScar}(x, y, z) \cdot 0), \text{CSadd}_3(x, y, z), w) \\
=^N \text{succ}^N(\text{CScar}_3(\text{succ}^N(\text{CScar}_3(x, y, z) \cdot 0), \text{CSadd}_3(x, y, z), w) \cdot 0 +^N \text{CSadd}_3(\text{succ}^N(\text{CScar}_3(x, y, z) \cdot 0), \text{CSadd}_3(x, y, z), w)) \cdot 0 \\
=^N \text{succ}^N(\text{succ}^N(\text{CScar}(x, y, z, w) +^N \text{CSadd}(x, y, z, w))) \cdot 0 \\
=^N \text{succ}^N(\text{succ}^N(x +^N y +^N z +^N w)) \cdot 0 \\
=^N \text{succ}^N(x +^N y) \cdot 0 +^N \text{succ}^N(z +^N w) \cdot 0 \\
=^N x_1 +^N y_1 +^N z_1 +^N w_1
\end{align*}
\]

Claim 3.2.88

1. \(\text{CARADD}(0x) =^N 0\)

2. \(\text{sum}(0x) = \text{CAR}(0x) +^N \text{ADD}(0x) =^N 0 +^N 0 =^N 0\)

Proof

1. By TIND on \(x\): \(\text{CARADD}(0\varepsilon) = \varepsilon =^N 0\), \(\text{CARADD}(0i) = 00 =^N 0\),

\[
\begin{align*}
\text{CARADD}(0x) &= \text{CScar}(\text{CAR}(0x \downarrow \downarrow), \text{ADD}(0x \downarrow \downarrow), \text{CAR}(\downarrow 0x), \text{ADD}(\downarrow 0x)) \cdot \\
&= \text{CSadd}(\text{CAR}(0x \downarrow \downarrow), \text{ADD}(0x \downarrow \downarrow), \text{CAR}(\downarrow 0x), \text{ADD}(\downarrow 0x)) \\
&=^N \text{CScar}(0, 0, 0, 0) \cdot \text{CSadd}(0, 0, 0, 0) \\
&=^N 0 \cdot 0 =^N 0.
\end{align*}
\]

2. Direct corollary of the first claim. \(\square\)

Theorem 3.2.25 \(\text{sum}(x) =^N \text{sum}(x \downarrow \downarrow) +^N \text{sum}(\uparrow x)\)
Proof

\[\text{sum}(x) = \text{CScar}(\text{CAR}(x \downarrow), \text{ADD}(x \downarrow), \text{CAR}(\uparrow x), \text{ADD}(\uparrow x)) + \text{N}\]
\[\text{CSadd}(\text{CAR}(x \downarrow), \text{ADD}(x \downarrow), \text{CAR}(\uparrow x), \text{ADD}(\uparrow x))\]

\[= \text{N} \text{CAR}(x \downarrow) + \text{N} \text{ADD}(x \downarrow) + \text{N} \text{CAR}(\uparrow x) + \text{N} \text{ADD}(\uparrow x)\]

\[= \text{sum}(x \downarrow) + \text{N} \text{sum}(\uparrow x) \quad \square\]

From this theorem, it is possible to prove that \(\text{sum}(xy) = \text{N} \text{sum}(x) + \text{N} \text{sum}(y)\) with a sequence of lemmas and theorems similar to the ones used to show that \(\text{AND}(xy) = \text{AND}(x) \land \text{B} \text{AND}(y)\). In particular, we have that \(\text{sum}(x0) = \text{N} \text{sum}(x) + \text{N} 0 = \text{N} \text{sum}(x)\) and \(\text{sum}(x1) = \text{N} \text{sum}(x) + \text{N} 1 = \text{N} \text{succ}(\text{sum}(x))\).

**Theorem 3.2.26**

\(\text{sum}(x) \leq \text{N} \text{sum}(1) = \text{N} |x|\)

**Proof**

By TIND on \(x\): \(\text{sum}(\varepsilon) = \text{sum}(1\varepsilon) = 0 = \text{N} |\varepsilon|\), \(\text{sum}(i) = \text{N} i \leq \text{N} 1 = \text{N} \text{sum}(1) = \text{N} |1|\), \(\text{sum}(x) = \text{N} \text{sum}(x \downarrow) + \text{N} \text{sum}(\uparrow x) \leq \text{N} \text{sum}(1 \downarrow) + \text{N} \text{sum}(1 \uparrow) = \text{N} \text{sum}(1) = \text{N} \text{sum}(1 \downarrow) + \text{N} \text{sum}(1 \uparrow) = \text{N} |x\downarrow| + \text{N} |x\uparrow| = \text{N} |x|\). To complete the inductive case for \(\text{sum}(1) = |x|\), we need to prove that \(|x| = \text{N} |x\downarrow| + \text{N} |x\uparrow|\) by TIND on \(x\): the base cases are trivial, and

\[|x\downarrow| + \text{N} |x\uparrow| = \text{N} x ?^\text{EL} ((|x\downarrow| + \text{N} |x\downarrow|, |x\downarrow| + \text{N} |x\downarrow| \cdot 1))\]

\[= \text{N} x ?^\text{EL} ((|x\downarrow| \cdot 0, |x\downarrow| + \text{N} \text{succ}(|x\downarrow|)))\]

\[= \text{N} x ?^\text{EL} ((|x\downarrow| \cdot 0, \text{succ}(|x\downarrow|) + \text{N} |x\downarrow|))\]

\[= \text{N} x ?^\text{EL} ((|x\downarrow| \cdot 0, |x\downarrow| \cdot 1)\]

\[= \text{N} |x|. \quad \square\]

**Lemmas for the proof of PHP**

**Lemma A.1**

\(\text{AND}(\text{or}^B_2(\text{not}^B(x), y)) \rightarrow^B \text{sum}(x) \leq \text{N} \text{sum}(y)\)

**Proof**

By Derived Rule 3.2.3:

\(\text{AND}(\text{or}^B_2(\text{not}^B(\varepsilon), y)) \rightarrow^B \text{sum}(\varepsilon) \leq \text{N} \text{sum}(y) = \text{AND}(y) \rightarrow^B \varepsilon \leq \text{N} \text{sum}(y) = 1,\)

\(\text{AND}(\text{or}^B_2(\text{not}^B(x), \varepsilon)) \rightarrow^B \text{sum}(x) \leq \text{N} \text{sum}(\varepsilon)\)

\[= \text{AND}(\text{not}^B(x)) \rightarrow^B \text{sum}(x) = \varepsilon = \neg^B \text{OR}(x) \rightarrow^B x = ^S 0 \cdot x = 1,\]

\(\text{AND}(\text{or}^B_2(\text{not}^B(xi), yj)) \rightarrow^B \text{sum}(xi) \leq \text{N} \text{sum}(yj)\)

\[= \text{AND}(\text{or}^B_2(\text{not}^B(x), y)) \land^B (\neg^B i \lor^B j) \rightarrow^B \text{sum}(x) + \text{N} i \leq \text{N} \text{sum}(y) + \text{N} j\]

\[= \text{AND}(\text{or}^B_2(\text{not}^B(x), y)) \land^B (\neg^B i \lor^B j) \rightarrow^B \text{sum}(x) \leq \text{N} \text{sum}(y) \land^B \neg^B (i \land^B \neg^B j) = 1. \quad \square\]
Lemma A.2 \[ \text{AND}(\text{or}_2^n(\text{not}^B(x), y)) \land^B \text{OR}(\text{and}_2^n(\text{not}^B(x), y)) \rightarrow \sum(x) <^N \sum(y) \]

Proof By Derived Rule [3.2.3]

\[
\begin{align*}
\text{AND}(\text{or}_2^n(\text{not}^B(x), y)) \land^B \text{OR}(\text{and}_2^n(\text{not}^B(x), y)) & \rightarrow B \sum(\varepsilon) <^N \sum(y) \\
= \text{AND}(y) \land^B \text{OR}(0y) & \rightarrow B \varepsilon <^N \sum(y) \\
= 0 \rightarrow^B 0 & = 1 \\
\text{AND}(\text{or}_2^n(\text{not}^B(x), \varepsilon)) \land^B \text{OR}(\text{and}_2^n(\text{not}^B(x), \varepsilon)) & \rightarrow B \sum(x) <^N \sum(\varepsilon) \\
= \text{AND}(\text{not}^B(x)) \land^B \text{OR}(0x) & \rightarrow B \sum(x) <^N \varepsilon \\
= 0 \rightarrow^B 0 & = 1 \\
\text{AND}(\text{or}_2^n(\text{not}^B(x), yj)) \land^B \text{OR}(\text{and}_2^n(\text{not}^B(x), yj)) & \rightarrow B \sum(xi) <^N \sum(yj) \\
= (\text{AND}(\text{or}_2^n(\text{not}^B(x), y))) \land^B (-^B i \land^B j) \land^B (\text{OR}(\text{and}_2^n(\text{not}^B(x), y)) \lor^B (-^B i \land^B j)) & \rightarrow^B \\
\sum(x) +^N i <^N \sum(y) +^N j
\end{align*}
\]

(and a simple check of all four cases for the possible values of \(i\) and \(j\) shows that the property holds in each one). \(\square\)

Lemma A.3 \[ \text{maskbit}(x, \text{first}_1(x)) = \text{OR}(x) \]

Proof By NIND on \(x\): \(\text{maskbit}(\varepsilon, \text{first}_1(\varepsilon)) = \text{markbit}(\varepsilon, \varepsilon) = \text{OR}(\text{and}_2^n(\varepsilon, \varepsilon)) = \text{OR}(\varepsilon)\), and \(\text{maskbit}(xi, \text{first}_1(xi)) = \text{maskbit}(x, \text{first}_1(x)) \lor^B (i \land^B \text{OR}(x) ?^B (0, i)) = \text{OR}(x) \lor^B \text{OR}(x) ?^B (0, i) = \text{OR}(x) ?^B (1, i) = \text{OR}(x) \lor^B i = \text{OR}(xi)\). \(\square\)

Lemma A.4

\(\text{(L)}\) \(\approx^B \text{lb}(x, y) = \text{maskbit}(x, (1 \cdot 0(y \triangleright x1))\varepsilon)\)

\(\text{(R)}\) \(\approx^B \text{rb}(x, y) = \text{maskbit}(x, (1 \cdot 0y)\varepsilon)\)

Proof (L) By Derived Rule [3.2.3]

\[
\begin{align*}
\text{maskbit}(\varepsilon, (1 \cdot 0(y \triangleright 1))\varepsilon) & \\
= y \approx^Z (\text{maskbit}(\varepsilon, 1), \text{maskbit}(\varepsilon, \varepsilon)) \\
= y \approx^Z (0, 0) = 0 = \approx^B \varepsilon = \approx^B \text{lb}(x, y),
\end{align*}
\]

\[
\begin{align*}
\text{maskbit}(x, (1 \cdot 0(\varepsilon \triangleright x1))\varepsilon) & \\
= \text{maskbit}(x, 1 \cdot 0x) \\
= \text{OR}(\text{and}_2^n(0x, 1 \cdot 0x)) \\
= (0 \land^B 1) \lor^B \text{OR}(\text{and}_2^n(x, 0x)) \\
= 0 \lor^B \text{OR}(0x) \\
= 0 = \approx^B \varepsilon = \approx^B \text{lb}(x, \varepsilon),
\end{align*}
\]

\(\square\)
Lemma A.5 \[ ^*{}\text{OR}(\text{delfirst}_1(x)) \land ^*{}\text{lb}(x, y) \rightarrow ^*{}\text{OR}(\text{lc}(x, y)) \]

Proof  By NIND on \( x: ^*{}\text{OR}(\text{delfirst}_1(\varepsilon)) \land ^*{}\text{lb}(\varepsilon, y) \rightarrow ^*{}\text{OR}(\text{lc}(\varepsilon, y)) = ^*{}\text{OR}(\varepsilon) \land ^*{}\varepsilon \rightarrow ^*{}\text{OR}(\varepsilon) = 0 \rightarrow ^*{}0 = 1 \), If \( y = \varepsilon \) or \( y >^L xi \), then \( \text{lb}(x, y) = \varepsilon \), which makes the antecedent of \( ^*{}\text{OR} \) false and the entire statement trivially true. Hence, we prove the inductive step under the implicit assumption that \( y \neq \varepsilon \) and \( y <^L xi \).

\[ ^*{}\text{OR}(\text{delfirst}_1(xi)) \land ^*{}\text{lb}(xi, y) \rightarrow ^*{}\text{OR}(\text{lc}(xi, y)) \]

\[ = y =^L xi \quad ^*{}\text{OR}(^*{}\text{OR}(x) ?^*{}(0, i)) \land ^*{}\text{lb}(x, y) \rightarrow ^*{}\text{OR}(\text{lc}(xi, x)), \]

\[ ^*{}\text{OR}(^*{}\text{OR}(x) ?^*{}(0, i)) \land ^*{}\text{lb}(x, y) \rightarrow ^*{}\text{OR}(\text{lc}(xi, x)) \]

\[ = y =^L xi \quad ^*{}\text{OR}(^*{}\text{OR}(x) ?^*{}(1, i)) \land ^*{}\text{lb}(x, y) \rightarrow ^*{}\text{OR}(\text{lc}(xi, x)), 1 \]

\[ = y =^L xi \quad ^*{}\text{OR}(^*{}\text{OR}(x) ?^*{}(i, i)) \land ^*{}\text{lb}(x, y) \rightarrow ^*{}\text{OR}(\text{lc}(xi, x)), 1 \]

\[ = y =^L xi \quad ^*{}\text{OR}(^*{}\text{OR}(x) ?^*{}(0, i)) \land ^*{}\text{lb}(x, y) \rightarrow ^*{}\text{OR}(\text{lc}(xi, x)), 1 \]

\[ = y =^L xi \quad (1, 1) = 1 \]
Bibliography

[1] Bill Allen. Arithmetizing uniform NC. *Annals of Pure and Applied Logic*, 53:1–50, 1991.

[2] Toshiyasu Arai. A bounded arithmetic AID for Frege system. Technical Report FI–CXT1998–003, The Fields Institute, April 1998.

[3] David A. Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC$^1$. *Journal of Computer and System Science*, 38:150–164, 1989.

[4] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

[5] Stephen Bloch. Functional characterizations of uniform log-depth and polylog-depth circuit families. In *Proceedings of IEEE 7th Annual Structure in Complexity Theory Conference*, pages 193–206, Boston, Massachusetts, June 1992.

[6] Stephen Bloch. Function-algebraic characterizations of log and polylog parallel time. *Computational Complexity*, 4:175–205, 1994.

[7] Samuel R. Buss. *Bounded Arithmetic*, volume 3 of *Studies in Proof Theory*. Bibliopolis, Naples, 1986.

[8] Samuel R. Buss. Polynomial size proofs of the propositional pigeonhole principle. *Journal of Symbolic Logic*, 52(4):916–927, December 1987.

[9] Samuel R. Buss. Propositional consistency proofs. *Annals of Pure and Applied Logic*, 52:3–29, 1991.

[10] Samuel R. Buss. Algorithms for boolean formula evaluation and for tree contraction. In P. Clote and J. Krajíček, editors, *Arithmetic, Proof Theory and Computational Complexity*, pages 96–115. Oxford University Press, 1993.

[11] Samuel R. Buss. Relating the bounded arithmetic and polynomial time hierarchies. Manuscript, November 1994.
[12] Peter Clote. Sequential, machine-independent characterizations of the parallel complexity classes ALOGTIME, AC$^k$, NC$^k$, and NC. In S. R. Buss and P. Scott, editors, *Feasible Mathematics*, pages 49–69. Birkhäuser, 1989.

[13] Peter Clote. ALOGTIME and a conjecture of S. A. Cook. *Annals of Mathematics and Artificial Intelligence*, 6:57–106, 1992. Extended abstract in: Proceedings of IEEE Symposium on Logic in Computer Science, Philadelphia, June 1990.

[14] Peter Clote. On polynomial size Frege proofs of certain combinatorial principles. In P. Clote and J. Krajíček, editors, *Arithmetic, Proof Theory and Computational Complexity*, pages 162–184. Oxford University Press, 1993.

[15] Peter Clote and Gaisi Takeuti. First order bounded arithmetic and small Boolean circuit complexity classes. In Peter Clote and Jeffrey Remmel, editors, *Feasible Mathematics II*, volume 13 of *Progress in Computer Science and Applied Logic*, pages 154–218, Boston, 1995. Birkhäuser.

[16] Alan Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Proceedings of the 1964 International Congress for Logic Methodology and the Philosophy of Science*, pages 24–30, Amsterdam, 1964. North Holland.

[17] Stephen Cook. Relating the provable collapse of $P$ to $NC^1$ and the power of logical theories. In Paul Beame and Samuel Buss, editors, *Proof Complexity and Feasible Arithmetics: DIMACS Workshop, April 21–24, 1996*, volume 39 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 73–91. American Mathematical Society, 1998.

[18] Stephen A. Cook. Feasible constructive proofs and the propositional calculus. In *Proceedings of the Seventh Annual ACM Symposium on the Theory of Computing*, pages 83–97, May 1975.

[19] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44(1):36–50, March 1979.

[20] Stephen A. Cook and Alasdair Urquhart. Functional interpretations of feasibly constructive arithmetic. *Annals of Pure and Applied Logic*, 63(2):103–200, September 1993.

[21] Russell Impagliazzo, Toniann Pitassi, and Alasdair Urquhart. Upper and lower bounds for tree-like cutting planes proofs. In *Proceedings of IEEE 9th Annual Symposium on Logic in Computer Science*, pages 220–228, 1994.
[22] Jan Johannsen. A bounded arithmetic theory for constant depth threshold circuits. In Petr Hájek, editor, GÖDEL '96, volume 6 of Springer Lecture Notes in Logic, pages 224–234, 1996.

[23] Jan Krajíček. Bounded Arithmetic, Propositional Logic, and Complexity Theory, volume 60 of Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1995.

[24] Jan Krajíček, Pavel Pudlák, and Gaisi Takeuti. Bounded arithmetic and the polynomial hierarchy. Annals of Pure and Applied Logic, 52:143–153, 1991.

[25] Daniel Leivant. A foundational delineation of computational feasibility. In Proceedings of IEEE 6th Annual Symposium on Logic in Computer Science, 1991.

[26] Daniel Leivant. Peano theories and their computable functions. Manuscript (extended abstract), December 1992.

[27] Alexander A. Razborov. Bounded arithmetic and lower bounds in boolean complexity. In Peter Clote and Jeffrey Remmel, editors, Feasible Mathematics II, volume 13 of Progress in Computer Science and Applied Logic, pages 344–386, Boston, 1995. Birkhäuser.

[28] Walter L. Ruzzo. On uniform circuit complexity. Journal of Computer and System Sciences, 22(3):365–383, June 1981.

[29] Gaisi Takeuti. Frege proof system and TNC^0. Manuscript, 1994.