Representing and computing with types in dynamically typed languages

Jim Newton

L'École Doctorale Informatique, Télécommunications et Électronique,

EDITE de Paris

Representing and Computing with Types in Dynamically Typed Languages

Extending Dynamic Language Expressivity to Accommodate Rationally Typed Sequences

Presented and defended to the public and before distinguished members of the jury:

Reporter	Pr. Robert Strandh	Université Bordeaux / LaBRI / UMR 5800
Reporter	Dr. Pascal Costanza	Imec
Jury member	Dr Barbara Jobstmann	École Polytechnique Fédérale de Lausanne (EPFL)
Jury member	Dr. Jean Bresson	IRCAM / CNRS / Sorbonne Université
Jury member	Dr. Chrisophe Rhodes	Goldsmiths University of London
Director	Pr. Thierry Géraud	EPITA/LRDE
Advisor	Dr Didier Verna	EPITA/LRDE

Jim Edward Newton

20 November 2018

Abstract

In this report, we present code generation techniques related to run-time type checking of heterogeneous sequences. Traditional regular expressions [HMU06, YD14] can be used to recognize well defined sets of character strings called *rational languages* or sometimes *regular languages*. Newton *et al.* [NDV16] present an extension whereby a dynamic language may recognize a well defined set of heterogeneous sequences, such as lists and vectors.

As with the analogous string matching regular expression theory, matching these *regular type expressions* can also be achieved by using a finite state machine (deterministic finite automata, DFA). Constructing such a DFA can be time consuming. The approach we chose, uses meta-programming to intervene at compile-time, generating efficient functions specific to each DFA, and allowing the compiler to further optimize the functions if possible. The functions are made available for use at run-time. Without this use of meta-programming, the program might otherwise be forced to construct the DFA at run-time. The excessively high cost of such a construction would likely far outweigh the time needed to match a string against the expression.

Our technique involves hooking into the Common Lisp type system via the `deftype` macro. The first time the compiler encounters a relevant type specifier, the appropriate DFA is created, which may be a $\Omega(2^n)$ operation, from which specific low-level code is generated to match that specific expression. Thereafter, when the type specifier is encountered again, the same pre-generated function can be used. The code generated is $\Theta(n)$ complexity at run-time.

A complication of this approach, which we explain in this report, is that to build the DFA we must calculate a disjoint type decomposition which is time consuming, and also leads to sub-optimal use of `typecase` in machine generated code. To handle complication, we use our own macro `optimized-typecase` in our machine generated code. Uses of this macro are also implicitly expanded at compile time. Our macro expansion uses BDDs (Binary Decision Diagrams) to optimize the `optimized-typecase` into low level code, maintaining the `typecase` semantics but eliminating redundant type checks. In the report we also describe an extension of BDDs to accomodate subtyping in the Common Lisp type system as well as an in-depth analysis of worst-case sizes of BDDs.

Résumé

Cette thése prèsente des techniques de gènèration de code liées à la vèrification dynamique de types de sèquences hètèrogénes mais règuliéres. Nous gènèralisons les expressions rationnelles classiques aux expressions rationnelles de type, en adaptant leur surface syntaxique, reprèsentation interne, calcul, optimisation, et sèrialisation (gènèration de code). Nous comparons la forme native basèe sur des S-Expressions avec une reprèsentation par Diagrammes de Dècision Binaire, enrichis pour reprèsenter les opèrations boolèennes dans un treillis de types prenant en charge le sous-typage. Nous introduisons alors la notion de Dècomposition Maximale en Types Disjoints, nous prouvons l'existence et l'unicitè d'une telle dècomposition, et nous explorons plusieurs solutions algorithmiques pour calculer celle-ci. La Dècomposition Maximale en Types Disjoints est utilisèe pour gènèrer un automate fini et dèterministe reprèsentant une expression rationnelle de type. Cet automate est utilisè à la compilation pour gènèrer à son tour du code de vèrification dynamique de type. Le code ainsi gènèrè ne tient a priori pas compte d'èventuelles redondances dans les vèrifications de type. Nous ètudions donc ces problémes plus en dètail et utilisons à nouveau la thèorie des Diagrammes de Dècicion Binaire afin d'èliminer les calculs redondants et de produire ainsi du code optimisè. Le travail prèsentè dans cette thése s'attache à traiter des problémes de représentations et de calcul sur des types dans les langages dynamiques.

Dedication

À Serge, en remerciement de son amour et de son soutien durant ces douze dernières années.

Il y a trois ans et demi, sous forme de boutade, il me disait: «ça ne serait pas mal d'être marié à un docteur»...

Sans lui, je ne me serais jamais lancé dans cette aventure...souvent excitante, parfois stressante, mais jamais insurmontable. Et cela je le dois à Serge qui a toujours su être un excellent soutien.

Acknowledgments

I want to express my appreciation to all the people who helped me in contribution to this PhD project. Unquestionable thanks go to Didier Verna and Theo Geraud for giving me the idea and the chance to pursue this PhD and for the many times they challenged my ideas leading to more thorough research and a better understanding of my research topic. In a literal sense, without Didier and Theo, this PhD project would never have been possible.

I would like to convey my esteemed gratitude to my steering committee, Robert Strandh and Carlos Argon for the frank feedback, valuable guidance, and insightful perspective they gave leading up to and during my yearly follow up reviews.

Sincere appreciation goes to Pascal Costanza for his help many times during this PhD project, for discussions, for reviewing articles in the very early, chaotic stages, and for generously agreeing to be an official reviewer of my thesis.

Thank you in anticipation to all the jury not already mentioned, for the valuable feedback on my thesis: Barbara Jobstmann, Christophe Rhodes, and Jean Bresson.

Due recognition goes to all of my colleagues at LRDE for their contributions in various ways to my research project: to Clément Démoulins for all his help with the LRDE compute cluster without which my computation would still be running; to Guillaume Tochon for his patient and enthusiastic help with many statistics related questions; to Alexandre Duret-Lutz and Maximilien Colange for our white board discussions, sometimes silly, but too many to recount; to Akim Demaille for his technical input, especially early on in the area of regular languages which turned out to have a significant influence on the technical direction of my research. Also thanks to Daniella Becker for her help with the French language and administration and so many other things without which I would have been lost so many times, and also to Daniella for so often lending a willing ear to hear my complaints.

Appreciation to Priti Singh for her meticulous proofreading of the manuscript and for suggesting many corrections in terms of grammar, clarity, vocabulary, and page formatting.

Thanks to several acquaintances on `comp.lang.lisp` for helping to resolve technical and historical questions: Pascal Bourguignon, Lieven Marchand, Barry Margolin, Kent M. Pitman, Doug Katzman, and Jeff Barnett. Thanks also to Jean Éric Pin, Yann Régis-Gianas for assisting me in resolving several technical issues. Thanks as well to Giuseppe Castagna for kindly helping me many times to gain perspective, for our conversations and for his publications, which were invaluable and inspiring, and also introducing me to Mariangiola Dezani who gladly provided me with more useful historical perspective and context.

And special thanks to Robert Strandh and Kathleen Callaway for their encouragement many times during the past three years of this interesting and enjoyable project.

Contents

Nomenclature

$[n, m]$	Integer interval	Notation 3.2, page 37.
$\mathcal{B}$	Index of the belt row of a worst-case ROBDD	Definition 6.9, page 136.
$\bigcap$	Unary intersection operator.	Definition 8.8, page 167.
$\bigcup$	Unary union operator.	Definition 8.7, page 167.
$\bigsqcup$	Mutually disjoint union.	Notation 8.60, page 180.
$\bot$	Lattice Bottom	Notation 2.6, page 21.
$\widetilde{V}$	Distributed complement	Definition 8.34, page 172.
$\Delta^{2M}\mathscr{H}_n(x)$	Histogram difference function	Definition 6.6, page 113.
$\parallel$	The set disjoint relation	Notation 2.2, page 20.
$\mathcal{E}_{rat}$	Rational expressions	Notation 3.22, page 41.
η	Boolean formula variable density.	Definition 5.23, page 100.
$\mathfrak{F}$	Distributed intersection	Definition 8.43, page 174.
${}^{M}\mathscr{H}_n(x)$	Sampled histogram function	Notation 6.3, page 108.
$\mathscr{H}_n(x)$	Histogram function	Definition 6.1, page 106.
$\overline{\mathscr{H}_n}(x)$	Normalized histogram function	Definition 6.2, page 108.
$\mathfrak{D}$	Disjunctive closure	Definition 8.33, page 171.
$\mathfrak{C}$	Conjunctive closure	Definition 8.32, page 171.
$\lceil x \rceil$	The ceiling function	Notation 6.28, page 143.
$\lfloor x \rfloor$	The floor function	Notation 6.27, page 143.
$\|\Delta^{M}\mathscr{H}_n\|_2$	L_2 norm of histogram difference function	Notation 6.7, page 113.
$\mathcal{L}_\Sigma$	Rational languages	Notation 3.20, page 40.
$\nparallel$	Not disjoint	Notation 2.3, page 20.
$\nu(r)$	nullable	Definition 3.26, page 43.
$\mathscr{P}(S)$	Set of pairs	Notation 6.41, page 151.
$\partial_w L$	Rational derivative	Definition 3.27, page 44.
$m^{\underline{2}}$	Number of order pairs of m items	Notation 6.12, page 137.
ϕ	Probability distribution function	Notation 6.5, page 112.
$\mathbb{P}(U)$	The power set of U.	Definition 8.11, page 167.
ψ	Real valued threshold function	Definition 6.26, page 143.
ρ_n	Residual compression ratio	Definition 6.8, page 134.

${}^{n}r_i$	Size of k'th row, viewed top down	Notation 6.17, page 139.
${}^{n}R_k$	Size of k'th row, views bottom up	Notation 6.13, page 138.
$[\![r]\!]$	Language generated by expression	Notation 3.21, page 41.
$\mathscr{B}(V)$	Sigma Algebra	Definition 9.1, page 183.
Σ	alphabet	Definition 3.1, page 37.
Σ^*	Set of all finite length words	Notation 3.8, page 38.
Σ^1	The set of single letter words	Notation 3.7, page 38.
σ_n	Standard deviation given a histogram	Notation 6.4, page 111.
${}^{n}S_k$	Sum on nodes in rows $k+1$ to n	Definition 6.14, page 138.
$\subset$	Strict subset or equal	Notation 2.5, page 21.
θ	The threshold function	Definition 6.25, page 142.
$\top$	Lattice Top	Notation 2.7, page 21.
$\lvert UOBDD_n \rvert$	Size of unreduced ordered BDD	Notation 5.3, page 88.
ε	empty word	Notation 3.5, page 38.
$\lVert f \rVert_\mu$	Average norm	Definition 10.6, page 214.
$\lVert f \rVert_1$	Averaged L_1 norm	Definition 10.7, page 215.
$\lVert f \rVert_{rms}$	Averaged RMS norm	Definition 10.8, page 215.
$\oplus$	Exclusive or, symmetric difference	Notation 5.6, page 91.
$A \cdot B$	Concatenation of languages	Definition 3.12, page 39.
A^*	Kleene closure of language A	Definition 3.18, page 40.
A^n	Language concatenated with itself multiple times.	Definition 3.14, page 39.
$node()$	ROBDD node constructor	Notation 6.39, page 151.
r^+	match a rational expression one or more times	Notation 3.29, page 45.
row_a^b	Nodes in a range of rows	Notation 6.40, page 151.
$u \cdot v$	Concatenation of words	Definition 3.10, page 38.
Z	Student score	Definition 10.9, page 215.
$A \to Y$	Function type	Definition 2.16, page 28.
congruent	Congruent BDD nodes	Definition 5.5, page 88.
DFA	Deterministic Finite Automaton	Definition 3.25, page 42.
language in Σ	A set of words	Definition 3.6, page 38.
Lisp function type	Lisp function type	Definition 2.20, page 29.
NDFA	Non-Deterministic Finite Automaton	Definition 3.24, page 42.
ordered BDD	Ordered Binary Decision Diagram	Definition 5.2, page 87.
Product size	Size of input times size of output	Definition 10.1, page 206.
Rational language		Definition 3.19, page 40.
symmetric	Symmetric BDD node	Definition 5.4, page 88.
type	Common Lisp type	Definition 2.1, page 20.
type specifier	An object which denotes a type.	Definition 2.8, page 21.
UOBDD	Unreduced Ordered Binary Decision Diagram	Definition 5.1, page 86.
word	a sequence of characters from an alphabet	Definition 3.3, page 37.

Part I

Regular Sequences in Common Lisp

Figure 1: Me, not yet thinking about whether I want to write a thesis.

This PhD research project takes a look at a particular problem in some modern dynamic programming languages. Many modern programming languages support a feature that allows sequences to be dynamic and heterogeneous, meaning that the types of the content may not be fully known until run-time, and different types of objects may be contained in the same sequence. We provide a way to make available to the compiler certain information not about the explicit types but rather about regular patterns of types within the sequences. Further, this report describes a problem that seems to have an elegant solution, but when this apparent solution is investigated, several hard problems are revealed. Herein, solutions to these problems are proposed and certain performance aspects of these problems are analyzed.

In Part I, we start in Chapter 1 with a high-level overview of the problem. In Chapter 4 we look at extending the Common Lisp type system to accommodate regular sequence types. Before diving into Chapter 4, we first introduce the Common Lisp type system in Chapter 2 and rational languages in Chapter 3.

Chapter 1

Overview

In this chapter, we summarize a technique for writing functions which recognize sequences of heterogeneous types in Common Lisp. The technique employs machine generated sequence recognition functions which are generated at compile time, and evaluated at run-time. The technique we demonstrate extends the Common Lisp type system, exploiting the theory of rational languages, BDDs (Binary Decision Diagrams), and the Turing complete macro facility of Common Lisp. The resulting system uses meta-programming to move a $\Omega(2^n)$ complexity[1] operation from run-time to a $\Omega(2^n)$ compile-time operation, leaving a highly optimized $\Theta(n)$ complexity[2] operation for run-time. We refer the reader to Wegener [Weg87, Section 1.5] for a discussion of Ω notation.

1.1 Introduction

There seems to be a trend that functional programming is becoming more popular, looking at the number of publications *e.g.*, about functional languages [Saj17, SGC15] and functional programming [Cuk17, Wam11]. This apparent rise may be due to the need for distributed computing. The functional paradigm provides tools for avoiding certain problems encountered in distributed algorithms. Scala Spark [KKWZ15] is an example of a powerful and popular functional interface for implementing Big Data computation. Even languages that are not purely functional can take advantage of a functional paradigm. Haveraaen *et al.* [HMR+15] describe the use of functional techniques in modern Fortran in high performance computing. Swaine [Swa17] illustrates blatant and subtle functional properties in several popular scripting languages like Clojure, Elixer, Scala, Swift, and Lua. Hughes [Hug89] reports that the modularity provided by functional languages enables programs to express results rather than computing by side effect, thus eliminating a source of bugs.

An introduction to functional programming can take any of various paths. For a seasoned Lisp programmer it makes sense to start by looking at the untyped lambda calculus [Chu41, Pie02] to understand the elegant model based on evaluation. Steele and Gabriel [SG93] suggest that Lisp languages are concerned more with expressiveness than anything else. Indeed, the expressiveness which Lisp in general offers is closely related to its underpinnings in the lambda calculus, *i.e.* the evaluation-based model and the use of functions to express composition of computation. However, to manage larger problems, the programmer needs more weaponry of abstraction. The Common Lisp [Ans94] programming language offers an arsenal of paradigms, enabling the programmer to use functional style when appropriate, and also enables object orientation [BDG+88, Kee89], meta object programming [Pae93, KdRB91], macro programming [Hoy08] and many others.

Another kind of abstraction which the student of functional languages encounters is the type system in the form of the typed lambda calculus [Pie02]. Types in programming languages bring at least three genres of capabilities to the respective languages, to different degrees depending on the programming language: (1) enabling the compiler to choose data representations and optimizations [THFF+17], to calculate allocation size, and to perform pointer arithmetic (like in the C language [KR88]); (2) detecting or eliminating certain programmer errors such as argument compatibility between definition and call-site (such as in gradually typed [CL17] functional languages and in C++ using generic programming paradigm [LGN12, LGN10]) allowing programmers to write type safe programs and allowing compilers to safely make certain optimizations, and (3) allowing the programmer to make run-time decisions based on dynamic types of run-time data.

The Common Lisp type system [Ans94, Section 4.2 Types] provides information to the compiler for memory allocation and space requirements, for detection of certain types of programmer errors, and for for certain optimizations. The type system may also be exploited to make run-time decisions based on type of information. This ability to make run-time decisions based on type of data adds expressive power to programs, and allows

[1] We use $\Omega(2^n)$ to mean that the complexity is bounded below by an exponential.
[2] We use $\Theta(n)$ to denote that the complexity is bounded above and below by a linear function.

the concept of type to be extended from a structural distinction to a semantic distinction. Castagna [CF05] reports that extending the concept of type to a semantic distinction enables the use of set theory on the type algebra. In fact, Common Lisp defines a type simply as a set of objects. Many kinds of run-time type checks incur penalties in terms of execution speed. Nevertheless, the cost of such run-time decisions can be mitigated by eliminating the redundant calculations or even moving such calculations to compile-time in certain cases.

In this research work, we have exploited the expressive ability of the Common Lisp programming language in terms of sequence types. This report describes the current state of research, including a high-level view of the problem of incorporating rational sequences into the Common Lisp type system, surface syntax, and conversion to internal representation. Certain issues of optimizing performance are also discussed concerning type checking calculations at run time. Some expensive calculations can be moved to compile time. The thesis focuses on compile time calculation, proving the correctness of algorithms, performance measurement of various approaches, and experimenting with different data structures.

This work will be beneficial to large number of users beyond the narrow audience of Lisp programmers. With the growing popularity of functional languages and even new Lisp dialects such as Clojure, different features of Common Lisp are explored in other languages. Certainly much of the historically unique power of Common Lisp comes from its dynamic nature, modern languages seek to access much of this flexibility in statically typed languages, which are sometimes disguised as gradually typed [CL17]. An example of such a language is Scala [OSV08, CB14]. This language attempts to make type declarations optional except in certain cases, allowing the programmer to write code which looks, in many cases, like a dynamic language. The compiler adds type declarations and assertions where needed, and thereafter it seeks to completely remove run-time type checks. We suggest that such languages may also benefit from some of the results of our work.

Recently, several languages have introduced tuple types (C++ [Str13, Jos12], Scala [OSV08, CB14], Rust [Bla15]). Our work provides similar capability of such tuple types for a dynamic programming language. The Shapeless [Che17] library allows Scala programmers to exploit the type-level programming capabilities through heterogeneous lists.

We also believe that as more work is done in programming languages which support heterogeneously typed tuples and sequences, especially in languages which also have type systems supporting set theoretical operations such as union and intersection [FCB08b, CF05], that the need will naturally arise to express patterns of types in those sequences.

1.2 Background

Common Lisp is a programming language defined by its specification [Ans94] and, with several implementations, including both open source and commercial. For the research explained in this report we have used SBCL [New15] as implementation of choice. All implementations share the common specified core of functionality, and each implementation extends that functionality in various ways.

Two Common Lisp features which we exploit are its macro system and its type system. The following is a very high-level summary of Common Lisp types and macros.

The Common Lisp macro system [Gra96, Section 10.2] allows programmers to write code which writes code. A macro may be defined using `defmacro` and such macros are normally *expanded* at compile time into code which is compiled and made available for execution at run time. Thanks to the homoiconicity [McI60, Kay69] of the Common Lisp language, macros take arguments which are Lisp objects (symbols, strings, numbers, lists *etc.*), and return program fragments which are also Lisp objects. Programmers writing macros may use any feature of the language in the macros themselves. There are several well understood caveats associated with Common Lisp macros. Costanza [CD10] explores the most notable of these—hygienic issues. Costanza extends the work of Clinger [CR91] who claims that Common Lisp suffers from problems that make it impossible to write macros that work correctly. Expert Common Lisp programmers understand these difficulties and restrictions and normally write macros exploiting the package system and the `gensym` function to avoid name conflicts.

The Common Lisp type system [Ans94, Section 4] can be understood by thinking of types as sets of objects. Subtypes correspond to subsets. Supertypes correspond to supersets. The empty type, called `nil`, corresponds to the empty set. The system comes with certain predefined types, such as `number`, `fixnum`, `rational` and many more. Additionally programmers may compose *type specifiers* which are syntax for expressions of types in terms of other types. These types are intended to be both human and machine readable. For example, `(or number string)` expresses the union of the set of numbers with the set of strings. Likewise type specifiers may use the operators `and`, `not`, `member`, and `satisfies` respectively for conjunction, complementation, enumeration, and definition by predicate.

Common Lisp allows types to be used in variable and function declaration, slot definitions within objects, and element definition with arrays. These declarations are normally considered during program compilation. In addition Common Lisp provides several other built-in macros and functions for type-based run-time reflection, *e.g.*, `typep`, `typecase`, `subtypep`, `check-type`. The programmer may associate new type names with composed

types by using `deftype` whose syntax is similar to that of `defmacro`.

1.3 Typed heterogeneous sequences

While the Common Lisp programmer can specify a homogeneous type for all the elements of a vector [Ans94, Section 15.1.2.2], or the type for a particular element of a list, [Ans94, System Class CONS], two notable limitations, which we address in this report, are 1) that there is no standard way to specify heterogeneous types for different elements of a vector, 2) neither is there a standard way to declare types (whether heterogeneous or homogeneous) for all the elements of a list.

A model for typing heterogeneous sequences is inspired from the theory of rational languages such as presented by Hopcroft [HMU06]. A rational language, which we formally define in Chapter 3 is a set of finite sequences which essentially obey certain language-specific pattern. A *rational expression* characterizes or specifies that pattern.

Example 1.1 (A rational expression denoting a rational language)**.** The rational expression $(a \cdot b^* \cdot c)$ defines the set of all sequences of characters which begin with the character 'a', end with the character 'c', and interpose zero or more (but finitely many) occurrences of character 'b'. Such sequences include the strings `"ac"`, `"abc"`, and `"abbbbc"`, but not the string `"bc"`. The set of all such strings is called the rational language of $(a \cdot b^* \cdot c)$.

In a programming language such as Common Lisp which supports arbitrary collections of objects into sequences, we can easily think of the types of those objects as obeying patterns similar to rational expressions. A *rational type expression*, addressed in Chapter 4, abstractly denotes the pattern of types within such sequences. The concept is envisioned to be intuitive to the programmer in that it is analogous to patterns described by regular expressions.

As addressed in Chapter 2, the Common Lisp language models types as sets. We can therefore think of rational languages not only as sets but equivalently as types.

Example 1.2 (The type associated with a rational type expression)**.** This example is completely analogous to Example 1.1.

The rational type expression $(string \cdot number^* \cdot symbol)$ denotes the set (and thus the type) of finite sequences which begin with an object of type `string`, end with an object of type `symbol`, and interpose zero or more (but finitely many) objects of type number. Such sequences include vectors like `#("hello" 1 2 3 world)` and lists like `("hello" world)`, but not the list `(3 four)`. The set of all such sequences is the type identified by the rational type expression $(string \cdot number^* \cdot symbol)$.

Rational expressions match character constituents strings according to a character equality predicate. By contrast, rational type expressions match elements of sequences by element type membership predicates.

A rational type expression is a mathematical expression composed of symbols, superscripts, and infix operators. To denote a rational type in the Common Lisp programming language, we abide by the Lisp tradition and define a surface syntax based on ASCII characters and prefix operators. This machine friendly s-expression based syntax, we call *regular type expression*.

We have integrated regular type expressions into the Common Lisp language by extending the type system with a user-defined type (see Section 4.4.2). We have implemented a parameterized type named `rte` (regular type expression), via `deftype`. The call-by-name argument of `rte` is a *regular type expression* whose grammar is detailed in Figure 4.1 in Section 4.2.1. The members of such type are all sequences matching the given regular type expression.

Example 1.3 (Example uses of the `rte` type specifier)**.**

```
(defclass C ()
  ((point :type (and list
                      (rte (:cat number number))))
   ...))

(defun F (X Y)
```

```
  (declare (type (rte (:* (cons number)))
                 Y))
  ...)

(typecase object
  ((rte (:* (:cat string (:* number) symbol))
   ...)
  ...))
```

As with all user defined types, the `rte` type can be used anywhere Common Lisp expects a type specifier. Example 1.3 illustrates some use cases. The `point` slot of the class `C`, is declared as being a list of exactly two numbers. The function `F` expects an argument, `Y`, which is a sequence of lists, where each list has a number as its first element.

1.4 Finite automata

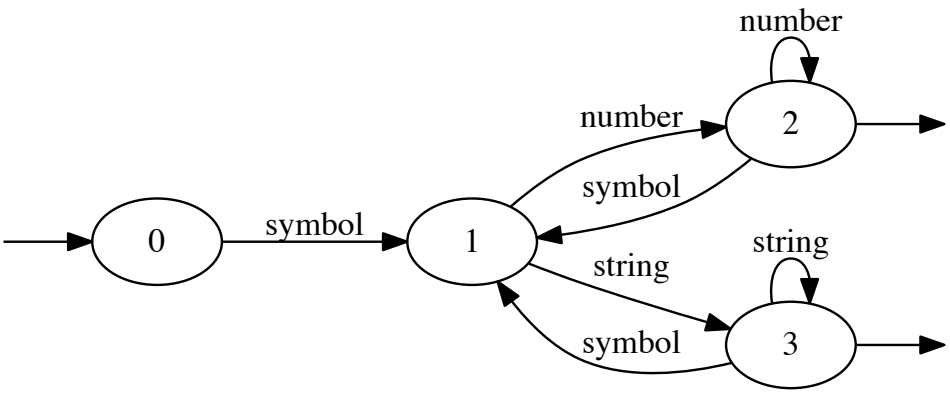


Figure 1.1: Finite state machine (DFA) (of rational type expression $(symbol \cdot (number^+ + string^+))^+$

In Section 1.3 we gave a high-level introduction to rational type expressions including how they can be used in a Common Lisp program to add expressivity to the type system. In this section, we take a look into the implementation. This implementation includes application of standard automata theory and the discovery of several interesting and challenging problems. The investigations and proposed solutions of these problems comprise the body of this report.

A standard mechanism for implementing rational expressions is with finite automata. The surface syntax, the regular expression, is parsed and converted into a finite state machine. This work must be done once per regular expression encountered in the program, typically with exponential complexity [Hro02]. Thereafter, a candidate string can be matched against a regular expression simply by *executing* the state machine, with linear complexity.

In implementing the rational type expressions, we follow this standard approach. The Common Lisp-friendly surface syntax is converted into an internal representation. This representation is converted directly into a deterministic finite automata (DFA), using a technique closely modeled after the Brzozowski rational derivative as described in Section 3.6.

We deviated from the Brzozowski rational derivative presented by Owens [ORT09a] because of the challenge posed by potentially overlapping types (see Section 4.4.6). In order to guarantee that the automaton be deterministic, we must decompose the set of types mentioned in a regular type expression into a larger set of non-intersecting types. This problem is referred to as Maximal Disjoint Type Decomposition (MDTD) and is discussed in Chapters 8 and 9.

1.5 Optimized code generation

Example 1.4 (Code generated to match regular type expression)**.**

```
(lambda (sequence)
  (tagbody
   L0
     (when (null sequence)
       (return nil)) ; rejecting
     (optimized-typecase (pop sequence)
       (symbol (go L1))
       (t (return nil)))
   L1
     (when (null sequence)
       (return nil)) ; rejecting
     (optimized-typecase (pop sequence)
       (number (go L2))
       (string (go L3))
       (t (return nil)))
   L2
     (when (null sequence)
       (return t)) ; accepting
     (optimized-typecase (pop sequence)
       (number (go L2))
       (symbol (go L1))
       (t (return nil)))
   L3
     (when (null sequence)
       (return t)) ; accepting
     (optimized-typecase (pop sequence)
       (string (go L3))
       (symbol (go L1))
       (t (return nil)))))
```

At runtime, determining whether a given sequence matches a rational type expression amounts to executing the automaton. There are two traditional ways of doing this. The first approach is to have a general purpose matching function which takes an automaton and candidate sequence as arguments. The function then executes the automaton as if it were a virtual machine, with instructions coming from the candidate sequence. The second approach, which we chose to use, is to compile the automaton directly into code on the target architecture, and execute that code at run-time to match a candidate sequence. In our case we compile the automaton into Common Lisp code, and allow the Common Lisp compiler to generate architecture-specific code. Example 1.4 shows what such Common Lisp code might look like.

One thing to note about the complexity of this function is that the number of states encountered when the function is applied to a sequence is equal to or less than the number of elements in the sequence. Thus the time complexity is linear in the number of elements of the sequence and is independent of the number of states in the DFA. Since we are able to generate this code at compile time (or load time in some cases) this means that whatever the complexity for generating the automaton is, the run-time complexity is linear.

Example 1.5 (Composed Common Lisp type specifier)**.**

```
(or (not number)
    (eql 42)
    (and fixnum (not unsigned))
    (and unsigned (not fixnum)))
```

In generating the code such as in Example 1.4, we treat each of the states in the automaton. Each state corresponds to a label in the resulting code. With each label there is an invocation of a macro named `optimized-typecase`, and each transition in the automaton corresponds to a clause in the `optimized-typecase`. Another challenging problem involved in this code generation is the question of which order to list these clauses. In some cases the types mentioned are highly composed involving union, intersection, and complement types, as illustrated in Example 1.5. Choosing the best execution order of the clauses could be important. We address this problem in Chapter 11, and in addressing this problem we explain our technique to transform code such as in Example 1.6

Example 1.6 (Example of `optimized-typecase` with intersecting types)**.**

```
(optimized-typecase object
  ((and unsigned
        (not (eql 42)))
   body-forms-1...)
  ((eql 42)
   body-forms-2...)
  ((and number
        (not (eql 42))
        (not fixnum))
   body-forms-3...)
  (fixnum
   body-forms-4...))
```

The technique is to convert such a `typecase` invocation into a type specifier, by substituting appropriate pseudo-predicates in place of the various body forms. For example the `typecase` in Figure 1.6 is converted to the type specifier in Figure 1.7.

Example 1.7 (Type specifier equivalent to `typecase` from Figure 1.6)**.**

```
(or (and (and unsigned (not (eql 42)))
         (satisfies P1))
    (and (eql 42)
         (not (and unsigned (not (eql 42))))
         (satisfies P2))
    (and (and number (not (eql 42)) (not fixnum))
         (not (and unsigned (not (eql 42))))
         (not (eql 42))
         (satisfies P3))
    (and fixnum
         (not (and unsigned (not (eql 42))))
         (not (eql 42))
         (not (and number
                   (not (eql 42))
                   (not fixnum)))
         (satisfies P4)))
```

After the type specifier as shown in Example 1.7 has been generated, we convert the type specifier into a binary decision diagram and thereafter into the Common Lisp code in Example 1.8 which is run-time equivalent to the `optimized-typecase` based code in Example 1.6.

Example 1.8 (Code expansion equivalent derived from Example 1.6)**.**

```
(lambda (object)
 (tagbody
   L1 (if (typep object 'fixnum)
          (go L2)
          (go L4))
   L2 (if (typep object 'unsigned-byte)
          (go L3)
          (go P4))
   L3 (if (typep object '(eql 42))
          (go P2)
```

```
        (go P1))
L4 (if (typep object 'number)
        (go L5)
        (return nil))
L5 (if (typep object 'unsigned-byte)
        (go P1)
        (go P3))
P1 (return (progn body-forms-1...))
P2 (return (progn body-forms-2...))
P3 (return (progn body-forms-3...))
P4 (return (progn body-forms-4...))))
```

1.6 Binary Decision Diagrams

A significant part of the work in manipulating Common Lisp types via BDDs was spent in investigation of BDDs themselves. Our work concentrated on a particular flavor of BDDs called ROBDD (Reduced Ordered Binary Decision Diagram). As mentioned already and as discussed in detail in Chapter 7, we intended the theory of BDDs to accomodate the Common Lisp type system. In Chapter 5 we present a pedagogical introduction to BDDs which should be instructional for the computer scientist unfamiliar with the subject. In that chapter, we also present some of our findings related to a Common Lisp implementation of ROBDDs, including Common Lisp implementations of certain ROBDD operations. We pay special attention to our technique for generating a randomly chosen ROBDD of n Boolean variables, from a space of size 2^{2^n}. These techniques enable our investigation of typical size and shapes of large ROBDDs in Chapter 6.

Chapter 6 focuses on a numerical and statistical analysis of certain size related properties of ROBDDs. In our investigation, we report the size and shape of typical and worst-case ROBDDs in relation to to the amount of memory allocation needed to represent them. We investigate size and shape, first through random sampling using techniques described earlier. This sampling seems to imply certain properties and tendencies about ROBDDs in general. Second, we follow that intuitive investigation by rigorously proving some of these intuitions. These derivations include formulas to predict worst-case size of an ROBDD of n Boolean variables. We introduce a concept called residual compression ratio which measures the storage efficiency of ROBDDs compared to raw truth table based storage. We follow this theoretical development with an explicit algorithm for generating a worst-case ROBDD for a given number of Boolean variables. We say "a worst-case ROBDD" because there are multiple ROBDDs which exhibit this worst-case size, and the algorithm makes this ambiguity explicit.

1.7 Source Available

All the source code presented in this report and the code used during the research is available as open source. It may be found and downloaded from the LRDE GitLab at `https://gitlab.lrde.epita.fr/jnewton/regular-type-expression`. The tagged commit from 14 October 2018 is `version-1.1.4`. The code has also been made available to quicklisp, but as of 14 october 2018, it was not yet available. The code is covered by the following license.

Chapter 2

Common Lisp Type System

In this chapter, we briefly summarize the Common Lisp type system including a definition of types and type specifiers. We give a high-level profile of some fundamental Common Lisp functions which operate on types at run-time. This chapter provides background information on the Common Lisp type system necessary to comprehend the theoretical and experimental applications explained in the chapters which follow.

An expressive feature of the Common Lisp type system is the ability to ask the question whether two specified types are in a subset relation. Although this question may be unanswerable in some situations, the fact that it is sometimes calculable, via the `subtypep` function, enables many of the other developments explained in this report. In particular:

1. The `subtypep` function is a critical component used in determinizing the finite state machine necessary for efficiently recognizing regular patterns in the types of heterogeneous sequences in Chapter 4.
2. The `subtypep` function is requisite in Chapter 7 where we extend Binary Decision Diagrams to model Common Lisp type specifiers.
3. And finally, the `subtypep` function is also indispensable in the kind of `typecase` optimization we explain in Chapter 11.

2.1 Types in Common Lisp

Definition 2.1. In Common Lisp, a *type* is a (possibly infinite) set of objects at a particular point of time during the execution of a program [Ans94].

In Common Lisp, types and functions may be redefined. Also an object of a particular class may be victim to the `change-class` function. Both of these situations as well as several others may cause a type to change its members while a program is running.

Notation 2.2. We use the symbol $\|$ to indicate the disjoint relation between sets. *I.e.*, we take $A \| B$ to mean $A \cap B = \varnothing$.

The notation, $\|$, is in no way standard notation. In fact we could not find any notation for this relation which is recognized by a majority of mathematicians. We chose this notation because it resembles the parallel relation of geometric lines. The mental image is that a line is a particular set of points, and parallel lines are such sets with no element (point) in common. Likewise, disjoint sets have no element in common.

Notation 2.3. We also say $A \nparallel B$ to mean $A \cap B \neq \varnothing$.

Example 2.4 (Disjoint and non-disjoint sets)**.** $\{1,3,5\} \parallel \{2,4\}$, and $\{1,3,5\} \nparallel \{2,3\}$.

Notation 2.5. We use the notation, $A \subset B$, $(A \supset B)$ to indicate that A is either a strict subset (superset) of B or is equal to B. When we wish to designate a strict subset (superset) relation, we denote $A \subsetneq B$ $(A \supsetneq B)$.

In type theory it is customary to use the symbols $\top$ and $\bot$ to represent respectively the supertype of all types and the subtype of all types.

Notation 2.6. The symbol, $\bot$, represents the Boolean false value, in a Boolean algebra context; the empty type, in a type system context; or the empty set, $\varnothing$, in a set theoretical context.

Notation 2.7. The symbol, $\top$, represents the Boolean true value, in a Boolean algebra context; the universal type, in a type system context; or the universal set, in a set theoretical context.

As illustrated in Figure 2.1, Common Lisp programmers may take many ideas about types from the intuition they already have from set algebra. Two given types might be intersecting such as the case of `unsigned-byte` and `fixnum` in the figure, (unsigned-byte $\nparallel$ fixnum). Types may be disjoint such as `float` and `fixnum`, (float $\parallel$ fixnum). Types may have a subtype relation such as `fixnum` and `number`, (fixnum $\subset$ number). Types may have more complicated relations such as $\big(\text{bit} \subset (\text{fixnum} \cap \text{unsigned-byte}) \subset \text{rational}\big)$.

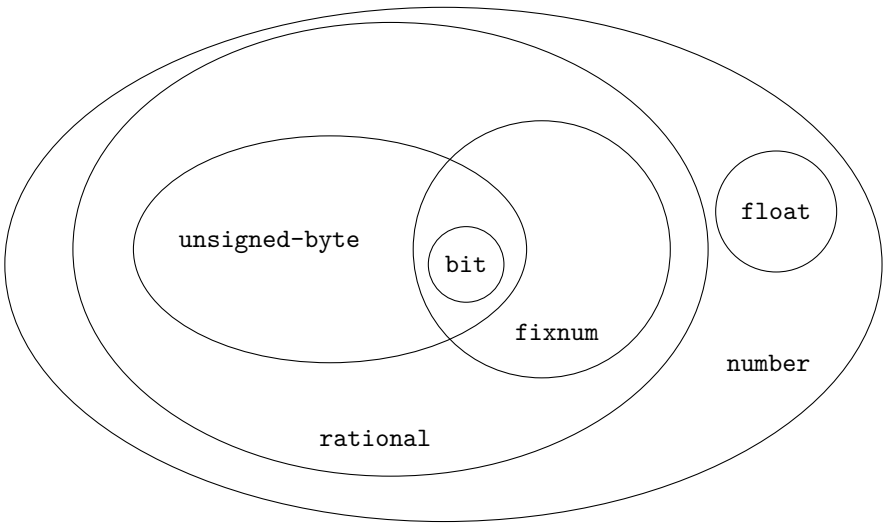


Figure 2.1: Some example Common Lisp types showing their intersection and subset relations

An object can belong to more than one type. Types are never explicitly represented as objects by Common Lisp. Instead, they are referred to indirectly by the use of *type specifiers*.

Definition 2.8. From the Common Lisp specification [Ans94], a *type specifier* is an expression that denotes a type.

Example 2.9 (Type specifiers)**.** Example taken from the Common Lisp specification [Ans94]. The symbol `random-state`, the list `(integer 3 5)`, the list `(and list (not null))`, and the class named `standard-class` are type specifiers.

A further discussion of type specifiers may be found in the specification [Ans94, Section 4.2.3].

New type specifiers can be defined using `deftype`, `defstruct`, `defclass`, and `define-condition`. But type specifiers indicating compositional types are often used on their own, such as in the expression `(typep x '(or string (eql 42)))`, which evaluates to `true` either if `x` is a string, or is the integer `42`.

Two important Common Lisp functions pertaining to types are `typep` and `subtypep`. The function `typep`, a set membership test, is used to determine whether a given object is of a given type. The function `subtypep`, a subset test, is used to determine whether a given type is a *recognizable* subtype of another given type. The function call `(subtypep T1 T2)` distinguishes three cases:

That `T1` is a subtype of `T2`,

That `T1` is not a subtype of `T2`, or

That subtype relationship cannot be determined.

Section 2.4 discusses situations for which the subtype relationship cannot be determined.

2.2 Semantics of type specifiers

There is some disagreement among experts as to how to interpret certain semantics of type specifiers in Common Lisp. To avoid confusion, we state explicitly our interpretation.

The situation that the user may specify a type such as `(and fixnum (satisfies evenp))` is particularly problematic, because the Common Lisp specification contains a dubious, non-conforming example in the specification of `satisfies`. The problematic example in the specification says that `(and integer (satisfies evenp))` is a type specifier and denotes the set of all even integers. This claim contradicts the Common Lisp specification in at least two ways.

Bourguignon [Bou17] explains the first violation that `evenp` is not a conforming argument of `satisfies`. The argument of `satisfies` must designate a predicate, which is a function (not a partial function) which returns a generalized Boolean as its first value. A function which may fail to return is not a predicate. In particular `(evenp nil)` does not return, but instead signals an error.

The second reason that `(and fixnum (satisfies evenp))` is non-conforming is that the specification of the `AND` type specifier states that `(and A B)` is the intersection of types `A` and `B` and is thus the same as `(and B A)`. This presents a problem, because `(typep 1.0 '(and fixnum (satisfies evenp)))` evaluates to `nil` while `(typep 1.0 '(and (satisfies evenp) fixnum))` signals an error. We implicitly assume, for optimization purposes, that `(and A B)` is the same as `(and B A)`. Specifically, we interpret the `AND` and `OR` types as being commutative with respect to their operands. Consequently, if certain type checks have side-effects (errors, conditions, changing of global state, IO, interaction with the debugger), then optimized code does not guarantee that those side-effects occur in the expected order, nor that they even occur at all. Therefore, in our treatment of types we consider that type checking with `typep` is side-effect free, and in particular that it never signals an error. This assumption allows us to reorder the checks as long as we do not change the semantics of the Boolean algebra of the `AND`, `OR`, and `NOT` specifiers.

Admittedly, that `typep` never signal an error is an assumption we make knowing that it may limit the usefulness of our results, especially since some non-conforming Common Lisp programs may happen to perform correctly absent our optimizations. That is to say, our optimizations may result in errors in some non-conforming Common Lisp programs. The specification clearly states that certain run-time calls to `typep` even with well-formed type specifiers must signal an error, such as if the type specifier is a list whose first element is `values` or `function`. Also, as mentioned above, an evaluation of `(typep object '(satisfies F))` will signal an error if `(F object)` signals an error. One might be tempted to interpret `(typep object '(satisfies F))` as `(ignore-errors (if (F object) t nil))`, but that would be a violation of the specification which is explicit that the form `(typep x '(satisfies p))` is equivalent to `(if (p x) t nil)`.

We assume, for this report, that no such problematic type specifier is used in the context of `typecase`.

2.3 Computation details regarding types

The Common Lisp language has flexible type calculus which makes type related computation possible by using human readable type specifiers. Even with certain limitations, s-expressions are an intuitive data structure for programmatic manipulation of type specifiers in analyzing and reasoning about types.

If `T1` and `T2` are Common Lisp type specifiers, then the type specifier `(and T1 T2)` designates the intersection of the types. Likewise `(and T1 (not T2))` and `(and (not T1) T2)` are the two type differences. Furthermore, the Common Lisp function `subtypep` can be used to decide whether two given types are equivalent or disjoint, and `nil` designates the empty type [Bak92]. See Figure 2.10 for definitions of the Common Lisp functions `type-intersection`, `type-relative-complement`, `types-disjoint-p`, and `types-equivalent-p`.

Implementation 2.10 (Definition of type calculus helper functions)**.**

```
(defun type-intersection (T1 T2)
  `(and ,T1 ,T2))

(defun type-relative-complement (T1 T2)
  `(and ,T1 (not ,T2)))

(defun type-null-p (T1)
  `(subtypep ,T1 nil))

(defun types-disjoint-p (T1 T2)
  (type-null-p (type-intersection T1 T2)))

(defun types-equivalent-p (T1 T2)
  (multiple-value-bind (T1<=T2 okT1T2) (subtypep T1 T2)
    (multiple-value-bind (T2<=T1 okT2T2) (subtypep T2 T1)
      (values (and T1<=T2 T2<=T1) (and okT1T2 okT2T2)))))
```

The function `types-disjoint-p` works because a type is empty if it is a subtype of `nil`. The function `types-equivalent-p` works because two types (sets) contain the same elements if each is a subtype (subset) of the other.

The definition of `types-equivalent-p` while correct is not written as efficiently as it could be. In particular there is no need to evaluate `(subtypep T2 T1)` in the case that `(subtypep T1 T2)` has already returned `nil`.

2.4 Unanswerable question of the subtype relation

There is an important caveat. The `subtypep` function is not always able to determine whether the named types have a subtype relationship or not [Bak92]. In such a case, `subtypep` returns `nil` as its second argument. This situation occurs most notably in the cases involving the `satisfies` type specifier. Consider Example 2.11 using the types `(satisfies evenp)` and `(satisfies oddp)`. The first problem we face is that an attempt to test type membership using such a predicate may be met with errors, such as when the argument of the `oddp` function is not an integer, as shown in Example 2.11.

Example 2.11 (Problems with `satisfies`.)**.**

```
(typep 1 '(satisfies oddp))
==> T
(typep 0 '(satisfies oddp))
==> NIL
(typep "hello" '(satisfies oddp))

The value "hello" is not of type INTEGER.
   [Condition of type TYPE-ERROR]
```

Even though usage of `satisfies` with the predicate `oddp` is in violation of the Common Lisp specification (according to our interpretation explained in Section 2.2), such usage seems to be fairly common in real-world Common Lisp programs. For example the Google corporate Common Lisp coding Style Guide[BF, Sec Pitfalls] recommends using the human readable form as shown in Example 2.12 as well as assuring the function `prime-number-p` "MUST accept objects of any type". Additionally, the Google style guide requires that the predicate be callable at compile time.

We also note here that the example in the Google corporate Style Guide violates the advise of Norvig and Pitman [NP93, p 13] in its usage of `when` as a two-branch expression.

Example 2.12 (Google Common Lisp Style Guide advice about `satisfies`.)**.**

```
(deftype prime-number () (satisfies prime-number-p)) ; Bad
(deftype prime-number () (and integer (satisfies prime-number-p)) ; Better

(eval-when (:compile-toplevel :load-toplevel :execute)
  (defun prime-number-p (n)
    (when (integerp n) ; Better
      (let ((m (abs n)))
        (if (<= m *prime-number-cutoff*)
            (small-prime-number-p m)
            (big-prime-number-p m))))))
```

Because of the error demonstrated in Example 2.11 it becomes a little easier if we define two types `odd` and `even` using `deftype`. Implementation 2.13 shows the initial type definitions which will be improved later. We see very quickly that the system has difficulty reasoning about these types.

Implementation 2.13 (Initial version of type definitions of `odd` and `even`)**.**

```
(deftype odd ()
  '(and integer (satisfies oddp)))
==> ODD

(deftype even ()
  '(and integer (satisfies evenp)))
==> EVEN

(subtypep 'odd 'even)
==> NIL, NIL

(subtypep 'odd 'string)
==> NIL, NIL
```

The `subtypep` function returns `nil` as its second value, indicating that SBCL is unable to determine whether `odd` is a subtype of `even`. Similarly, SBCL is not able to determine that `odd` is not a subtype of `string`. This behavior is conforming. According to the Common Lisp specification, the `subtypep` function is permitted to return the values `false` and `false` (among other reasons) when at least one argument involves type specifier `satisfies` [Ans94].

SBCL cannot know whether two arbitrary functions might return `true` given the same argument. The human can see that the types `odd` and `even` are non-empty and disjoint, and thus neither is a subtype of the other.

Notice also that `(subtypep 'odd 'string)` returns `nil as second value` indicating that it was unable to determine whether `odd` is a subtype of `string`. At a glance it would seem that `(and integer (satisfies oddp))` is definitely not a subset of `string`, because `(and integer (satisfies oddp))` is a subset of `integer` which is disjoint from `string`. But there's a catch. The system does not know that `odd` and `even` are non-empty. If a type `A` is empty, then in fact `(and integer A)` is a subtype of `string` because $integer \cap A = \varnothing \subset string$ [Kat15].

2.5 Type specifier manipulation

Common Lisp programs which manipulate type specifiers have traditionally used s-expressions as the programmatic representations of types, as described in the Common Lisp specification [Ans94, Section 4.2.3]. Such choice of internal data structure offers advantages such as homoiconicity [McI60, Kay69], making the internal representation human readable in simple cases, and making programmatic manipulation intuitive, as well as enabling the direct use of built-in Common Lisp functions such as `typep` and `subtypep`. However, this approach presents some challenges. Such programs often make use of ad-hoc logic reducers—attempting to convert types to canonical form. These reducers can be complicated and difficult to debug. In addition run-time decisions about type equivalence and subtyping can suffer performance problems as Baker [Bak92] explains.

The Maximal Disjoint Type Decomposition (MDTD) will be introduced in Chapter 8. To correctly implement the MDTD by either strategy described in Chapter 9, we need operators to test for type-equality, type disjoint-ness, subtype-ness, and type-emptiness. Given a *subtype* predicate, the other predicates can be constructed. The emptiness check: $A = \varnothing \iff A \subset \varnothing$; the disjoint check: $A \parallel B \iff A \cap B \subset \varnothing$; type equivalence $A = B \iff A \subset B$ and $B \subset A$.

As another example of how the Common Lisp programmer might manipulate s-expression based type specifiers, consider the following problem. In SBCL 1.3.0, the expression `(subtypep '(member :x :y) 'keyword)` returns `nil,nil`, rather than `t,t`. Although this is conforming behavior, the result is unsatisfying, because clearly both `:x` and `:y` are elements of the `keyword` type. The function defined in Implementation 2.14 manipulates the given type specifier s-expressions to augment the built-in version of `subtypep` to better handle this particular case. Regrettably, the user cannot force the system to use this smarter version internally.

Implementation 2.14 (`smarter-subtypep`)**.**

```
(defun smarter-subtypep (t1 t2)
  (multiple-value-bind (T1<=T2 OK) (subtypep t1 t2)
    (cond
      (OK
       (values T1<=T2 t))
      ;; (eql object) or (member object1 ...)
      ((typep t1 '(cons (member eql member)))
       (values (every #'(lambda (object)
                          (typep object t2))
                      (cdr t1))
               t))
      (t
       (values nil nil)))))
```

As mentioned above, programs manipulating s-expression based type specifiers can easily compose type intersections, unions, and relative complements as part of reasoning algorithms. Consequently, the resulting programmatically computed type specifiers may become deeply nested, resulting in type specifiers which may be confusing in terms of human readability and debuggability. The programmatically generated type specifier shown in Example 2.15 is semantically correct for programmatic use, but confusing if it appears in an error message, or if the developer encounters it while debugging.

Example 2.15 (Programmatically generated type specifier)**.**

```
(or
 (or (and (and number (not bignum))
          (not (or fixnum (or bit (eql -1)))))
     (and (and (and number (not bignum))
               (not (or fixnum (or bit (eql -1)))))
          (not (or fixnum (or bit (eql -1))))))
 (and (and (and number (not bignum))
           (not (or fixnum (or bit (eql -1)))))
      (not (or fixnum (or bit (eql -1))))))
```

This somewhat obfuscated type specifier is semantically equivalent to the more humanly readable form `(and number (not integer))`. Moreover, it is possible to write a Common Lisp function to *simplify* many complex type specifiers to simpler form.

There is a second reason apart from human readability which motivates reduction of type specifiers to canonical form. The problem arises when we wish to programmatically determine whether two s-expressions specify the same type, or in particular when a given type specifier specifies the `nil` type. Sometimes this question can be answered by calls to `subtypep` as in `(and (subtypep T1 T2) (subtypep T2 T1))`. However, as mentioned earlier, `subtypep` is allowed to return `nil,nil` in some situations, rendering this approach futile in many cases. If, on the other hand, two type specifiers can be reduced to the same canonical form, we can conclude that the specified types are equal.

2.6 Type reduction using s-expressions

As mentioned in Section 2.5, our implementations of the MDTD algorithm depend heavily on calculating type intersections and relative complements. We found that s-expression based solutions performed poorly when they were naïvely coded using the `type-intersection` and `type-relative-complement` functions in Implementation 2.10. The problem is that as a rule of thumb, the size of the type specifier grows exponentially on each call to `type-intersection` or `type-relative-complement`. We determined that some amount of simplification was necessary as part of `type-intersection` and `type-relative-complement`.

We have implemented such a type simplification function, `reduce-lisp-type`. It does a good job of reducing the given type specifier toward a canonical form. The strategy of the function is to recursively descend the type specifier s-expression, re-writing sub-expressions, as shown in Figure 2.2. This recursive process repeats until a fixed point is reached, the fixed point being a type specifier in a disjunctive normal form, *i.e.* an OR of ANDs such as `(or (and (not a) b) (and a b (not c)))`. The simplification procedure follows the models presented by Sussman and Abelson [AS96, p. 108] and Norvig [Nor92, ch. 8].

We use the idiom toward a canonical form because this algorithm does not always succeed in reducing two equivalent type specifiers to equal s-expressions. For example, the operations in Figure 2.2 will not rewrite the type specifiers `(or A B)` and `(or B A)` to a common form, nor will they simplify `(or (and A B E) (not E))` and `(or (and A B) (not E))` to a common form. Either of these two pairs of equivalent types could be simplified further by enhancing the `reduce-lisp-type` function to handle more cases in its algebraic manipulation. However, no amount of symbolic manipulation would be able to simplify `(and string fixnum)` to `nil` nor to reduce `(or number fixnum)` to `number`. To recognize these kinds of expressions, the simplification function would have to exploit the `subtypep` function. In particular it would need to perform $O(n^2)$ searches on every occurrence of `and` and `or` to find pairs which have disjoint or subset relations. As Baker [Bak92] predicts, this can be slow.

Input expression	Output expression
`(and A (and B C))`	`(and A B C)`
`(or A (or B C))`	`(or A B C)`
`(and A nil)`	`nil`
`(and A B t)`	`(and A B)`
`(and A B C (not B))`	`nil`
`(and A)`	`A`
`(and)`	`t`
`(or A t)`	`t`
`(or A B nil)`	`(or A B)`
`(or A)`	`A`
`(or)`	`nil`
`(or A B C (not B))`	`t`
`(and A (or B C) D)`	`(or (and A B D) (and A C D))`
`(not t)`	`nil`
`(not nil)`	`t`
`(not (not A))`	`A`
`(not (or A B))`	`(and (not A) (not B))`
`(not (and A B))`	`(or (not A) (not B))`

Figure 2.2: Some s-expression based operations for simplifying a Common Lisp type specifier.

We found, not surprisingly, that s-expression based type specifier rewriting is a computationally intensive operation, not only theoretically but also in practice. The problem seems to be exacerbated in several ways: first, by the fact that given a type specifier, our implementation of `reduce-lisp-type` cannot know whether it has already been reduced. Figuring out whether it needs to be reduced, is as complex as actually reducing it.

The second factor which seems to slow the computation of simplification is that `subtypep` sometimes returns `nil` as second argument (as explained in Section 2.4), indicating the subtype relation was not determined. The consequence is that the simplification function is forced to assume the worst case and maintain redundant information in the type specifier. Each such additional component increases the computation time quadratically, at least locally for the type specifier being considered.

Still a third reason it is difficult to implement a good s-expression based type simplification function is that optimizations in the function for one Common Lisp implementation, such as SBCL, may turn out to be pessimizations for another Common Lisp implementation, such as Allegro CL. We do not mean this as a reproach against any implementation or vendor, but it seems that micro-optimizations at the level of the application programmer, may have unintended and surprising performance penalties when later trying to make code portable.

As mentioned above, optimizations can be made in our algorithm resulting in better canonicalization, computation, or allocation. However, each such optimization risks making the code more difficult to understand and maintain. We have also observed cases where optimizations which speed the computation in SBCL have a pessimization effect in other Lisp implementations. One advantage of the ROBDD data structure, which we present in Chapter 5, is that it maintains canonical form by design, so an ROBDD need never be reduced once it has been allocated.

In any such s-expression based type reduction, one must find a balance between how much computation time is dedicated to the task *vs.* how good a reduction is needed.

The Common Lisp code implementing the s-expression based type reduction function can be found in Appendix A.

2.7 Function types

In Chapters 4, 7, 9, 10, and 11 we discuss certain computations based on Common Lisp type specifiers. Each of these computations depends on the ability to consistently determine whether there is a subset relation between two specified types, and moreover, to make that determination based on the syntax of the type specifiers and the rules of the type system. In each of the chapters, we ignore the question of the subset relation between subtypes of the `function` type. This area needs additional research. In this section (Section 2.7) we explain some of our findings which should be considered in any such future research.

1. Function type specifiers do not explicitly describe sets of values, but rather valid code transformations (Section 2.7.2).
2. Function type specifiers in Common Lisp seem to violate the Induced Subtype Rule (ISR) (Section 2.7.3).
3. The Common Lisp specification does not explicitly explain how to compute subset relations between function types (Section 2.7.6).
4. Common Lisp programs are forbidden from making certain run-time function subtype checks (Section 2.7.7).

2.7.1 Semantics of function types

Many approaches to type theory assign semantics to what we sometimes call arrow types, denoted $A \to Y$. The symbol $A \to Y$ carries the intuition of the set of functions mapping domain A to range Y, but the specifics vary depending on the particular type-theoretical approach taken. As we address in Section 2.7.2, the meaning of arrow types in Common Lisp is different from that in other common treatments.

Pierce [Pie02, Ch 9] defines the arrow type formally with a syntax and rewriting (evaluation) semantics, such as

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 \,.\, t_2 : T_1 \to T_2} \quad \text{[Abstraction]} \tag{2.1}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \; t_2 : T_{12}} \quad \text{[Application]}\,. \tag{2.2}$$

The meaning of these algebraic symbols is beyond the scope of this report, but suffice it to say that it defines exactly in which contexts arrow types may appear in a program, and exactly what effects they have on program evaluation. That is, rules (2.1) and (2.2) define exactly how arrow types behave in the type system of Pierce without relying on any intuition.

Castagna *et al.* [FCB08a, Sec 2.6] summarize the constraints of defining arrow types consistently. In a one to one interview, when posed the question, "What does $A \to Y$ mean?" Dr. Castagna responded, "It means

whatever you want it to mean for the type system you are trying to define, as long as you are consistent." He continued by giving an example of on-going work in which he defines arrow types for a lazy language[ACPZ18] in which certain terms may diverge as long as they are never evaluated.

Definition 2.16. If A and Y denote types, we take $A \to Y$ to mean the set of all unary functions mapping every element of type A to an element of type Y.

If we take Definition 2.16 to be the definition of arrow types, then we explicitly define the type to exclude functions which diverge or loop forever, even though it is impossible to distinguish the two programmatically. The code in Example 2.17 defines such a function, which is not in *integer* → *integer* according to Definition 2.16, because there exist integers, namely 0 and 1, for which the function fails to return an integer.

Example 2.17 (Function with divergent behavior)**.**

```
(defun strange-function (x)
  (declare (type integer x))
  (case x
   ((0)
    (loop :while t)) ; loop forever
   ((1)
    (error "some error"))
   (t
    (1+ x))))
```

2.7.2 Function types in Common Lisp

In our research into the MDTD problem, we make heavy use of the `subtypep` function. It is natural to ask under which conditions there is a subtype relation between two given arrow types. Common Lisp provides two mechanisms for specifying subtypes of `function`. The first is by using the Metaobject Protocol (MOP) [Pae93, KdRB91, Cos]. With this approach, we use the object system, CLOS, to define classes which inherit from the `function` class, and simultaneously from other mix-in classes to extend the behavior of function objects. When function subclasses are defined this way, Common Lisp updates the type lattice with new types corresponding to these classes, which behave as one would expect with regard to `subtypep`. An example is provided in Appendix E.

The second way the Common Lisp type system provides for describing subtypes of `function` is with a particular type specifier syntax. This syntax specifies a subtype of `function` which depends on the types of parameters and return value. An s-expression such as `(function (arg-types) return-type)` designates a subtype of `function`. Although the Common Lisp syntax allows specifying multiple arguments as well as optional arguments and multiple return values, it is sufficient for our purpose to limit our discussion to unary functions with a single return value.

The explanation of the function type in the Common Lisp specification is vague. Rather than a clear statement of what `(function (A) Y)` means in terms of types `A` and `Y`, [Ans94, Section FUNCTION] gives an explanation which is neither a necessary condition nor a sufficient condition for a function to be an element of such a type:

Claim 2.18. *Every element of type* `(function (A) Y)` *is a function that accepts arguments of type* A *and returns values of type* Y*.*

The specification further explains that if F has type `(function (A) Y)`, the consequences are undefined if F is called with an element not of type A. The specification does not indicate exactly what "a function that accepts arguments of type A" means; *i.e.* it is unclear whether this means that only elements of A are valid arguments of the function, that every element of A is valid argument of the function, or that some elements

of A are valid arguments of the function.[1] However, it does clarify that if the argument is not in A, then the result is not guaranteed to be in Y.

Rather than directly defining the semantics of the type `(function (A) Y)`, the specification says:

Claim 2.19 (Call-site rewriting rule)**.**
If a function `F` is declared of type `(function (A) Y)`, then any call such as `(F x)` may be replaced with the transformed code `(the Y (F (the A x)))`.

Because of Claim 2.19, we infer Definition 2.20, which we did not find anywhere explicitly stated in the Common Lisp specification.

Definition 2.20. The type `(function (A) Y)` is the largest set of functions which has the following property: if F is in the set, then every call site of the form `(F x)` can be replaced with `(the Y (F (the A x)))`, without changing the evaluation of the call site.

The Common Lisp specification (still in [Ans94, Section FUNCTION]) states that an `ftype` declaration for a function describes calls to the function, not the actual definition of the function. The exact set-theoretical properties of `calls to functions` are still unclear, and as we will discuss in Section 2.7.6, the definition was also up for debate during the Common Lisp specification process. For this reason (and others which we describe in the following sections), we omit considerations of function types from the type specifier discussions in the following chapters.

2.7.3 The Induced Subtype Rule (ISR)

The subtyping consequence, Corollary 2.21 follows from Definition 2.16.

Corollary 2.21. *If arrow types are defined as in Definition 2.16, then whenever $B \subset A$, we have $(A \to Y) \subset (B \to Y)$.*

Proof. We can see this by taking an $f \in A \to Y$, and showing that necessarily $f \in B \to Y$.

Suppose $B \subset A$, and take $\beta \in B$. Since all elements of B are in A, we have $\beta \in A$ so $(f\beta) \in Y$. □

Section 2.7.5 explains that this consequence does not apply to the Common Lisp type system. This discrepancy is mentioned in the X3J13 cleanup issue FUNCTION-TYPE-ARGUMENT-TYPE--SEMANTICS:RESTRICTIVE [vR88] that function argument type semantics are different from what users expect. One might wonder what would happen if we considered function type specifiers as behaving like other Common Lisp types specifiers with regard to set semantics. Under such an assumption, we would be able to reason about the semantics of such types using the semantics of sets. What we find, however, is that some familiar intuitions fail. In this section (Section 2.7.3) we see that a principle known as ISR (Claim 2.22) fails, and in Section 2.7.6 we see that certain intuitions regarding type intersection fail.

Claim 2.22 (Induced subtype rule (ISR))**.**
If $A \subset B$ and $X \subset Y$, then $B \to X$ is a subtype of $A \to Y$, denoted $(B \to X) \subset (A \to Y)$.

Claim 2.22 is not actually a principle supported in Common Lisp rather it is an assumption some users make as is further discussed in Section 2.7.8. The claim, which is a generalization of Corollary 2.21, is basically saying that function types are (should be) covariant with respect to return type and contravariant with respect to argument type. Castagna [Cas16, Section 2.2] provides a proof of this claim and for a weaker version of its converse.

Since function type specifiers are valid arguments to the `subtypep` function, we can test Claim 2.22 using SBCL 1.4.10 as in Examples 2.23 and 2.24. In Example 2.23, we see results which agree with the ISR. We see

[1] A restaurant may accept credit cards, but may at the same time not accept American Express.

that since $fixnum \subset number$, then `subtypep` considers `(function (T) fixnum)` to be a subtype of `(function (T) number)`, and `(function (T) number)` not to be a subtype of `(function (T) fixnum)`.

Example 2.23 (Function type specifiers with covariant return types)**.**

```
CL-USER> (subtypep '(function (t) fixnum) '(function (t) number))
T
T

CL-USER> (subtypep '(function (t) number) '(function (t) fixnum))
NIL
T
```

In contrast to Example 2.23, in Example 2.24 we see that `subtypep` does not respect ISR with respect to contravariance of function arguments. The `subtype` function, at least in SBCL, believes $(fixnum \to \top) \subset (number \to \top)$ and $(number \to \top) \not\subset (fixnum \to \top)$.

Example 2.24 (Function type specifiers with contravariant argument types)**.**

```
CL-USER> (subtypep '(function (number) t) '(function (fixnum) t))
NIL
T

CL-USER> (subtypep '(function (fixnum) t) '(function (number) t))
T
T
```

2.7.4 Degenerate function types

In this section, we take a brief look at arrow types composed of $\top$ and $\bot$. We see that Common Lisp, in particular SBCL, supports these degenerate arrow types even thought they seem to violate the definition of arrow types from Definition 2.20. First we derive a property of types $\top \to \bot$ and $\bot \to \top$ in Corollary 2.26, then test the predictions in Example 2.27.

A consequence of Claim 2.22 is shown in Theorem 2.25, and we can test the predictions of the theorem as shown in Example 2.27.

Theorem 2.25. *Given types A, B, X, and Y,*

$$\big((A \cup B) \to (X \cap Y)\big) \subset (A \to Y) \subset \big((A \cap B) \to (X \cup Y)\big)\,.$$

Proof. Since $A \subset A \cup B$ and $Y \cap X \subset Y$ by ISR we can conclude that

$$((A \cup B) \to (X \cap Y)) \subset (A \to Y)\,.$$

Since $A \cap B \subset A$ and $Y \subset X \cup Y$, then by ISR we can conclude that

$$(A \to Y) \subset \big((A \cap B) \to (X \cup Y)\big)\,.$$

So together we have

$$\big((A \cup B) \to (X \cap Y)\big) \subset (A \to Y) \subset \big((A \cap B) \to (X \cup Y)\big)\,.$$

□

In this section we take a brief look at the degenerate cases of Theorem 2.25.

Corollary 2.26. $(\top \to \bot) \subset (\bot \to \top)$.

Proof. Let $B = \neg A$ and $X = \neg Y$, and apply Theorem 2.25. □

In Example 2.27 we test Corollary 2.26. We see that SBCL does not respect the ISR. According to SBCL $(\top \to \bot)$ is not a subtype of $(\bot \to \top)$. In fact neither is a subtype of the other.

Example 2.27 (Testing Corollary 2.26 in SBCL)**.**

```
CL-USER> (subtypep '(function (t) nil) '(function (nil) t))
NIL
T

CL-USER> (subtypep '(function (nil) t) '(function (t) nil))
NIL
T
```

The correspondence of $\top \to \bot$ and $\bot \to \top$ to Common Lisp types is difficult to grasp. We see in the Example 2.28, that neither `(function (nil) t)` nor `(function (t) nil)` is empty as neither is a subtype of `nil`.

Example 2.28.

```
CL-USER> (subtypep '(function (nil) t) nil)
NIL
T

CL-USER> (subtypep '(function (nil) t) t)
T
T

CL-USER> (subtypep '(function (t) nil) t)
T
T

CL-USER> (subtypep '(function (t) nil) nil)
NIL
T
```

We further see in Example 2.29 that the two types, `(function (nil) t)` and `(function (t) nil)`, are non-disjoint (*i.e.* intersecting) types.

Example 2.29.

```
CL-USER> (subtypep '(and (function (nil) t) (function (t) nil)) nil)
NIL
T
```

By Definition 2.20, `(function (nil) t)` is the set of functions F such that `(F x)` can be replaced everywhere with `(F (the nil x))`. Likewise, `(function (t) nil)`, is the set of functions G such that `(G x)` can be replaced with `(the nil (G x))`. According to the `subtypep` implementation in SBCL, those are two distinct, intersecting, non-empty sets of functions. Exactly which two sets of functions are specified by these type specifiers is not clear.

Before leaving the discussion of degenerate arrow types in Common Lisp, we look briefly at $\bot \to \bot$ and $\top \to \top$. Example 2.30 shows that in Common Lisp we have $\bot \to \bot \subsetneq \top \to \top$, a strict subset relation.

Example 2.30 (`subtypep` with degenerate arrow types)**.**

```
CL-USER> (subtypep '(function (nil) nil) '(function (t) t))
T
T

CL-USER> (subtypep '(function (t) t) '(function (nil) nil))
NIL
T

CL-USER> (subtypep  '(function (nil) nil) '(function (t) nil))
T
T

CL-USER> (subtypep  '(function (t) nil) '(function (nil) nil))
NIL
T
```

The strict subset relation between $\bot \to \bot$ and $\top \to \top$ makes sense according to Definition 2.20. `(function (t) t)` is the set of unary functions which either diverge or not, *i.e.* the set of functions F for which `(F x)` can be replaced by `(the t (F (the t x)))`; while `(function (nil) nil)` contains only functions which diverge, *i.e.* `(F x)` can be replaced with `(the nil (F (the nil x)))`. It is, however curious to note that `(function (nil) nil)` does not contain all divergent unary functions, because, as Example 2.30 also shows, $\bot \to \bot \subsetneq \top \to \bot$, *i.e.* a strict subset relation.

2.7.5 Intuition of function type intersection

The examples and results shown in Sections 2.7.3 and 2.7.4 may come as a surprise to some readers, as they seem in violation of ISR. The theory of semantic subtyping [FCB08b] is a branch of computer science which attempts to put consistent semantics on types and subtypes to intuitively align with set theoretical semantics. As we explain in more detail in Section 2.8, at the time Common Lisp was being specified, the theory of semantic subtyping was not yet well understood. The Common Lisp specification does not explain explicitly how the `subtypep` function should behave given function type specifiers.

To illustrate the failing intuition of intersection, consider the `stringify` function in Example 2.31.

Example 2.31 (Stringify function)**.**

```
(defun stringify (x)
  (format nil "~A" x))
```

The function `stringify` certainly maps all ratios to string. According to Claim 2.18 we might be tempted to think that the function `stringify` is an element of the type `(function (ratio) string)`, and since the function also maps integers to strings, we might be tempted to think that it is an element of type `(function (integer) string)`. Furthermore, since the function is in both types, it must be an element of the intersection of the two types. Such an interpretation of Claim 2.18 would be erroneous according to the call-site rewriting rule (Claim 2.19). If the function `stringify` were a member of `(function (ratio) string)` then we would be able to rewrite any call-site, even `(stringify 3.1416)`, by the call-site rewriting rule. But we know that

`(stringify 3.1416)` is not equivalent to `(the string (stringify (the ratio 3.1416)))`, as `3.1416` is not an element of the type `ratio`.

Similarly, staying with our erroneous intuition, if a function is in the intersection of the two types: `(function (ratio) string)` and `(function (integer) string)`, *i.e.*, if that function maps both ratios and integers to strings, then that function maps all `(or ratio integer)` to strings.[2] From this logic, and in keeping with the intuitions of semantic type theory, one would conclude that the intersection of the two types was simply `(function ((or ratio integer)) integer)` = `(function (rational) integer)`. One would infer the rule $(A \to Y) \cap (B \to Y) = A \cup B \to Y$. However, as explained in Section 2.7.6, the Common Lisp type system works differently in relation to function type intersection.

2.7.6 Calculation of function subtype relation is underspecified

Section **System Class FUNCTION** of the Common Lisp specification does not specify how to compute the intersection of two function types of the same arity, but it does specify what rule the compiler may apply whenever a function is simultaneously declared to be in two function types. We state the rule restricted to unary function with a single return value. If the two type declarations for function `F` are in effect:

```
(function (A) X)
(function (B) Y)
```

then within the shared scope of the declarations, calls to `F` can be treated as if `F` were declared as the following:

```
(function ((and A B)) (and X Y))
```

We restate here the Common Lisp definition of function type intersection using arrow notation.

Claim 2.32 (Intersection induced subtype relation)**.** *If A, B, X, and Y are types, then the following relation holds with regard to intersections:*

$$(A \to X) \cap (B \to Y) \subset (A \cap B) \to (X \cap Y)$$

Doel [hd] points out that the two are not in explicit contradiction. However, the intersection in Claim 2.32 is different from what we would guess from the ISR (Claim 2.22), which says that $(A \to Y) \cap (B \to Y) \subset (A \cup B) \to Y$. We also find no description in the Common Lisp specification for answering subtype questions about unions and complements of functions or functions of unions and complements.

2.7.7 Run-time type check of functions considered harmful

At run-time a program is allowed to use `typep` to check whether an object is of type `function`, but only in the most simple form. An expression such as `(typep object 'function)` is permitted. However, the specification explicitly forces any run-time call to `typep` to signal an error as in Example 2.33, if a function type such as `(function () t)` is given.

Example 2.33 (Error signaled if using `function` type at run-time)**.**

```
CL-USER> (typep (lambda () 42) 'function)
T

CL-USER> (typep (lambda () 42) '(function () fixnum))

Function types are not a legal argument to TYPEP:
  (FUNCTION NIL (VALUES FIXNUM &REST T))
   [Condition of type SIMPLE-ERROR]
```

On the contrary, it is not clear from the specification what should occur with a run-time call to `subtypep` with the list form of a `function` type specifier. The specification contains what some people might interpret as contradictory information. It says, "The list form of the function type-specifier can be used only for declaration

[2] In Common Lisp the name given to `(or ratio integer)` is `rational`.

and not for discrimination." A call to `subtypep` is arguably neither declaration nor discrimination, at least not object discrimination. Even so, a run-time call to `subtypep` is not a declaration. We might reasonably conclude, based on that statement, that `subtypep` is not allowed to be called with the list form of a `function` type specifier, but that conclusion would be premature.

On the other hand, the specification for `subtypep` has a non-definitive clarification. It says "`subtypep` is permitted to return the values false and false only when at least one argument involves one of these type specifiers: `and`, `eql`, the list form of `function`, `member`, `not`, `or`, `satisfies`, or `values`." Even though only when does not mean if, we nevertheless interpret this statement to mean that a call to `subtypep` with `function` type specifiers is allowed.

Although we would have liked the MDTD algorithms to be applicable to function types, we currently consider this problem beyond the scope of the Common Lisp `subtypep` function for the reasons explained in Sections 2.7.3, 2.7.6, and 2.7.7. The problem of these inconsistencies seems to imply that programs should not rely on the return value of `subtypep` when dealing with function types. We thus ignore function types in the rest of this report.

It would seem that at least one of the contributors to the X3J13 cleanup issue, FUNCTION-TYPE--ARGUMENT-TYPE-SEMANTICS:RESTRICTIVE [vR88] shared our view, at least to some extent. We find a remark in the final paragraph of the cleanup issue, that the notion of "subtype" does not make sense in conjunction with the list form of function types because it is different from most type specifiers.

2.7.8 A historical perspective

It would appear that the architects of Common Lisp were not oblivious to the question of whether ISR should apply to Common Lisp function types. As mentioned briefly above, there is an X3J13 cleanup issue named FUNCTION-TYPE-ARGUMENT-TYPE-SEMANTICS:RESTRICTIVE [vR88]. This issue discusses the problem that the function argument type semantics are different from what users expect. It is mentioned that CLtL [Ste90, pp 47-18] requires the `cons` function be type `(function (t t) cons)` and also of type `(function (float string) list)`. If this requirement is interpreted this according to ISR, then the author is suggesting that `(function (float string) list)` is a subtype of `(function (t t) cons)` with covariant return value, $list \subset cons$; but contravariant arguments, $float \subset \top$ and $string \subset \top$.

According to the accepted Common Lisp specification (rather than that proposed in the cleanup issue), the `cons` function is not of type `(function (float string) list)`, because `(cons t nil)` is a valid call to `cons` but `(the list (cons (the float t) (the string nil)))` is not.

There is also a discussion in the X3J13 email archive [Gab90, Bar87] initiated by Jeff Barnett where he suggests and explains the ISR (of course without using that name) in what he called "PROBLEM II". His argument is that if $T_1 \subset T_2 \subset T_3$, then $(T_3 \to T_1) \subset (T_2 \to T_2)$ and $(T_1 \to T_3) \not\subset (T_2 \to T_2)$. Barnett demands that the `subtypep` function "must **reverse** the order of its arguments when checking argument types—original order is used when value types are checked." His reasoning was that most Common Lisp implementations included incremental compilation. And if a function is edited and recompiled in a way which narrowed its type (new type is a subtype of previous type) the compiler need not warn about call-sites. However, if the function type was changed in any other way, call sites might or might not be invalid.

Barnett's comments were motivated [Bar18] by experience with the CRISP [BP74, Bar10, Bar09] language from the early 1970s which was part of a speech understanding systems in which run time efficiency was highly important. The CRISP language was specified to be a strongly typed Lisp dialect, which defined types and in particular function types in a way compatible with what we now call ISR. The CRISP language defined a type [BP74, p. 49] as a collection of objects.

Unfortunately, the only follow-up response to "PROBLEM II" was by Nick Gall [Gab90, Gal87], who made a dubious claim which was never challenged. He claimed "The type specifier (FUNCTION ...) is not acceptable to TYPEP (pg. 47), therefore it is not acceptable to SUBTYPEP (pg. 72)." According to an interview with Nick Gall, he clarified that page 47 and 72 are referring to CLtL [Ste84] edition 1 (not 2). Contrary to the Gall's claim, however, we could not find anywhere in the Common Lisp specification where run-time calls to `subtypep` are prohibited.

2.8 Related work

Tobin *et al.* [THFF+17] argue that during the 1990s and later, developers preferred languages such as Lisp, Ruby, Python, and Perl because the statically typed languages which existed at the time lacked the flexibility they needed. The authors present a system for migrating programs incrementally by declaring types within Racket programs which were previously implemented exploiting the flexibility of a language lacking type declarations.

Castagna *et al.* [CL17] argue as well that many programmers prefer the flexibility and development speed of dynamically typed languages, but that, nevertheless, compilers may infer types from the program if certain

language constructs exist. The authors explore gradual typing in laguages which support set-theoretical types– languages whose types support union, intersection, and complementation. In particular they argue that adding such set theoretical operations to a type system facilitates a smooth transition from dynamic typing to static typing while given even more control to the programmer.

Around the time Common Lisp was under development, other programming language researchers were experimenting with set semantics for types. We have not been able to determine how much of this research influenced Common Lisp development or vice versa. Dunfield [Dun12] discusses works related to intersection types, but omits explicit references to the Common Lisp type system. He does mention the Forsythe [Rey96] programming language developed in the late 1980s, which provides intersection types, but not union types. Dunfield claims that intersections were originally developed in 1981 by Coppo *et al.* [CDCV], and in 1980 by Pottinger [Pot80] who acknowledges discussions with Coppo and refers to Pottinger's paper as forthcoming. Work on union types began later, in 1984 by MacQueen *et al.* [MPS84].

As in Common Lisp, languages which support intersection and union types [CL17, CF05, Pea17], should also be consistent with respect to subtype relations. Frisch *et al.* [FCB08b] referred to this concept as semantic subtyping. In particular, the union of types A and B should be a supertype of both A and B, the intersection of types A and B should be a subtype of both A and B, and if A is a subtype of B, then the complement of B should be a subtype of the complement of A. While Coppo and Dezani [CD80] discuss intersection types in 1980, according to a private conversation with one of the authors, Mariangiola Dezani, theoretical work on negation types originates in Castagna's semantic subtyping.

The theory of intersection types seems to have some influence on modern programming language extensions, at least in Java and Scala. Bettini *et al.* [BBDC+18] (including Dezani, mentioned above) discuss the connection of Java λ-expressions to intersection types.

The Scala language [OSV08] supports a type called `Either` which serves much of the purpose of a union type. `Either[A,B]` is a composed type, but has no subtype relation neither to A nor to B. Sabin [Sab11] discusses user level extensions to the Scala type system to support intersection and union types. Scala-3 [Ami16, RA16, dot18] promises to fully support intersection and union types in a type lattice, but not complementary types.

As in Common Lisp, which supports sequences of heterogeneously typed objects, several languages have started to introduce tuple types (C++ [Str13, Jos12], Scala [OSV08, CB14], Rust [Bla15]). Our work provides similar capability of such tuple types for a dynamic programming language. The Shapeless [Che17] library allows Scala programmers to exploit the type-level programming capabilities through heterogeneous lists.

We commented in Section 2.6 that when `subtypep` is unable to determine the subtype relation, the consequence is a quadratic growth in computation time of the type specifier simplification function. The work of Valais [Val18] may be useful in addressing this problem. Even if the Baker `subtypep` algorithm which Valais has implemented is slower than `cl:subtypep` by a linear factor, the fact that it never returns `nil` as second argument beyond the case of `satisfies`, could potentially improve the performance of type reduction.

In Section 2.7 we gave special attention to questions about the `(function (...) ...)` type in Common Lisp, with particular attention to the ISR. The section explains some of the historical perspectives which lead us to believe that the Common Lisp specification does not respect[3] ISR. The specification, in our opinion, should either have an explicitly defined way to calculate the subtype relation for function types, or it should prohibit their usage in conjunction with `subtypep`. We realize our opinion may be controversial. The opinion of Bourguignon and others may be found in [New18]. Finally, the Scala language [OSV08, RL17], allows programmers to decide within function and class declaration, which parameters are covariant, contravariant, or invariant.

2.9 Perspectives

For a better historical perspective, we would like to continue the investigation of the origins of the Common Lisp type system. We see several ideas in the Common Lisp type system which were areas of research outside the Lisp community during the time period that Common Lisp was being developed. However, we have not been able to find references between the two research communities. For instance Common Lisp defines type as a set of objects; similarly Hosoya and Pierce [Hos00, HP01] simplify their model of semantic subtyping by modeling a type as a set of values of the language.

We discovered quite late in this project that a bottom-up approach to converting a Common Lisp type specifier to DNF form was in many cases much better performing than the fixed point approach discussed in Section 2.6. The code for this approach, `type-to-dnf-bottom-up` is also shown in Appendix A after the code for `type-to-dnf`. Without this alternative bottom-up approach we found that some of our test cases never finished in Allegro CL, and with this additional optimization the Allegro CL performance was very competitive with

[3]RESPECT. RIP Aretha Louise Franklin, (March 1942 to August 2018) whose passing coincides closely with the submission of this thesis manuscript. She improved the world around her with her music, her activism, and her grace. Likewise, may our actions make the world a better place.

SBCL. On the other hand, we found that some of the performance tests in SBCL (as discussed in Chapter 10) exhausted the memory and we landed in the LDB (low level SBCL debugger). We believe there is merit in developing a correct, portable, high performance type simplifier; however we believe this is a matter where more research and experimentation are needed.

Chapter 3

Rational Languages

In Chapter 2 we discussed the Common Lisp type system, concentrating primarily on types of atomic values. We would like to introduce, in Chapter 4, a way to use the Common Lisp type system to describe sequences of objects whose types follow regular patterns. Before we can do that we describe, in the current chapter, what regular patterns are. We introduce the theory of rational languages which formalizes these patterns. The theory provides notation for describing the patterns mathematically as well as programmatically. The theory further provides an algorithm which uses finite state machines to convert the programmatic syntax into a decision algorithm for efficiently recognizing appropriate sequences of characters.

In the current chapter, we present character based regular expressions so that in Chapter 4 we may extend them to accommodate type sequences in Common Lisp.

3.1 Theory of rational languages

Definition 3.1. An *alphabet* is defined as any finite set, the elements of which are defined as *letters*. We generally denote the letters of an alphabet by Latin letter symbols. *E.g.*, $\Sigma = \{a, b, c\}$.

Notation 3.2. An Integer interval from n to m including the boundaries.

$$[n, m] = \{x \in \mathbb{Z} \mid n \leqslant x \leqslant m\}$$

Definition 3.3. Given an alphabet, Σ, a *word* of length $n \in \mathbb{N}$ is a sequence of n characters from the alphabet. This can be denoted as a function, mapping the set $[1, n] \to \Sigma$. We denote the sequence in an intuitive way, simply as a juxtaposed sequence of characters.

Example 3.4 (A rational expression as function on an integer interval)**.** For example *aabc* denotes the following function $aabc$: $[1, 4] \to \Sigma$

$$aabc(n) = \begin{cases} a & \text{if } n = 1 \\ a & \text{if } n = 2 \\ b & \text{if } n = 3 \\ c & \text{if } n = 4 \end{cases}$$

Notation 3.5. There is a word of zero length, called the *empty word*. It is denoted ε. The empty word is indeed a function $\varepsilon : \varnothing \subset \mathbb{N} \to \Sigma$.

Definition 3.6. A *language* is defined as a set of words; more specifically a *language in* Σ is a set of words each of whose letters are in Σ.

Notation 3.7. The set of all words of length one whose letters come from Σ is denoted Σ^1.

Notation 3.8. The set of all possible words of finite length made up exclusively of letters from Σ is denoted Σ^*.

Example 3.9 (Languages of an alphabet)**.** Examples of languages of $\Sigma = \{a, b\}$ are $\varnothing$, Σ, $\{a, aa, aaa, aaaa\}$, and $\{\varepsilon, ab, aaba, ababbb, aaaababababbb\}$.

Definition 3.10. If A, and B are a languages with alphabet Σ, and $a \in A$, $b \in B$, then we define the *concatenation* of a and b as the sequence of letters comprising a followed immediately by the sequence of characters comprising b. Such a concatenation of words is denoted either by a juxtaposition of symbols or using the $\cdot$ operator: i.e., ab or equivalently $a \cdot b$.

Precisely, if $a\colon [1, \ell_a] \to \Sigma$ and $b\colon [1, \ell_b] \to \Sigma$, then $u \cdot v\colon [1, (\ell_a + \ell_b)] \to \Sigma$, such that:

$$(a \cdot b)(n) = \begin{cases} a(n) & \text{if } 1 \leqslant n \leqslant \ell_a \\ b(n - \ell_a) & \text{if } \ell_a + 1 \leqslant n \leqslant \ell_a + \ell_b \end{cases} .$$

Theorem 3.11. *The concatenation operation defined in Definition 3.10 is associative.*

Proof. Let A, B, and C be languages. Take

$$\begin{aligned} a \in A \text{ with } a\colon [1, \ell_a] &\to \Sigma \\ b \in B \text{ with } b\colon [1, \ell_b] &\to \Sigma \\ c \in C \text{ with } c\colon [1, \ell_c] &\to \Sigma\,. \end{aligned}$$

Then we have,

$$\begin{aligned} \big((a \cdot b) \cdot c\big)(n) &= \begin{cases} (a \cdot b)(n) & \text{if } 1 \leqslant n \leqslant \ell_a + \ell_b \\ c(n - \ell_a - \ell_b) & \text{if } \ell_a + \ell_b + 1 \leqslant n \leqslant \ell_a + \ell_b + \ell_c \end{cases} \\ &= \begin{cases} a(n) & \text{if } 1 \leqslant n \leqslant \ell_a \\ b(n - \ell_a) & \text{if } \ell_a + 1 \leqslant n \leqslant \ell_a + \ell_b \\ c(n - \ell_a - \ell_b) & \text{if } \ell_a + \ell_b + 1 \leqslant n \leqslant \ell_a + \ell_b + \ell_c \end{cases} \\ &= \begin{cases} a(n) & \text{if } 1 \leqslant n \leqslant \ell_a \\ (b \cdot c)(n - \ell_a - \ell_b) & \text{if } \ell_a \ell_b + 1 \leqslant n \leqslant \ell_a + \ell_b + \ell_c \end{cases} \\ &= \big(a \cdot (b \cdot c)\big)(n) \end{aligned}$$

□

Definition 3.12. If A and B are languages, then $A \cdot B = \{u \cdot v \mid u \in A \text{ and } v \in B\}$. As a special case, if $A = B$, we denote $A \cdot A = A^2$. When it is unambiguous, we sometimes denote $A \cdot B$ simply as AB.

Theorem 3.13. *The language concatenation operator is associative.*
If A, B, and C are languages with alphabet Σ, then

$$(A \cdot B) \cdot C = A \cdot (B \cdot C)\,.$$

Proof. Take any word $w \in \big((A \cdot B) \cdot C\big)$. w can be expressed in the form $((a \cdot b) \cdot c)$ where $a \in A$, $b \in B$, and $c \in C$. By Definition 3.3, there exist ℓ_a, ℓ_b, $\ell_c \in \mathbb{N}$ such that $a\colon [1, \ell_a] \to \Sigma$, $b\colon [1, \ell_b] \to \Sigma$, and $c\colon [1, \ell_c] \to \Sigma$. By Theorem 3.11,

$$\big((a \cdot b) \cdot c\big)(n) = \big(a \cdot (b \cdot c)\big)(n) \in \big(A \cdot (B \cdot C)\big)\,.$$

So $\big((A \cdot B) \cdot C\big) \subset \big(A \cdot (B \cdot C)\big)$.

Now take $w \in \big(A \cdot (B \cdot C)\big)$. w can be expressed in the form $(a \cdot (b \cdot c))$ where $a \in A$, $b \in B$, and $c \in C$. By Definition 3.3, there exist ℓ_a, ℓ_b, $\ell_c \in \mathbb{N}$ such that $a\colon [1, \ell_a] \to \Sigma$, $b\colon [1, \ell_b] \to \Sigma$, and $c\colon [1, \ell_c] \to \Sigma$. By Theorem 3.11,

$$\big(a \cdot (b \cdot c)\big)(n) = \big((a \cdot b) \cdot c\big)(n) \in \big((A \cdot B) \cdot C\big)\,.$$

So $\big((A \cdot B) \cdot C\big) \supset \big(A \cdot (B \cdot C)\big)$. □

Definition 3.14. If A is a language, and $n \geqslant 0$ is an integer, then A^n is defined as follows:

$$A^n = \begin{cases} \{\varepsilon\} & \text{if } n = 0 \\ A \cdot A^{n-1} & \text{if } n > 0 \end{cases}\,.$$

Now we effectively have two definitions for A^2; *i.e.* $A^2 = A \cdot A$, and also $A^2 = A \cdot A^1$. Corollary 3.17 shows that these two definitions define the same set.

Theorem 3.15. *If $p = n + m$, then $A^p = A^n \cdot A^m$.*

Proof. By induction on p:
If $p = 0$, then $n = m = 0$. $A^0 = \{\varepsilon\} = \{\varepsilon \cdot \varepsilon\} = \{\varepsilon\} \cdot \{\varepsilon\} = A^0 \cdot A^0$.
Assume true for $p = k$, and show true for $p = k + 1$. *I.e.*, assume that if $k = m + n$ then $A^k = A^n \cdot A^m$.
Take $m + n = k + 1$, then $m + n - 1 = k$, so

$$\begin{aligned} A^{m+n} &= A \cdot A^{m+n-1} && \text{by Definition 3.14} \\ &= A \cdot A^{(m-1)+n} \\ &= A \cdot (A^{m-1} \cdot A^n) && \text{by inductive assumption} \\ &= (A \cdot A^{m-1}) \cdot A^n && \text{by Theorem 3.13} \\ &= A^m \cdot A^n\,. && \text{by Definition 3.14} \end{aligned}$$

□

Corollary 3.16.

$$A^m \cdot A^n = A^n \cdot A^m$$

Proof. This follows from Theorem 3.15.

$$A^m \cdot A^n = A^{m+n} = A^{n+m} = A^n \cdot A^m .$$

□

Corollary 3.17.

$$A^1 = A$$

Proof. By Definition 3.14 $A^1 = A \cdot A^0$.

Take $w_1 \in A^1$. So there exists $w \in A$ and $w_0 \in A^0$ such that $w_1 = w \cdot w_0$. But the only element in A^0 is ε, making $w_0 = \epsilon$. $w_1 = w \cdot \varepsilon = w \in A$. So $A^1 \subset A$.

Now, take $w \in A$. Since $w = w \cdot \varepsilon$, and $\varepsilon \in A^0 = \{\varepsilon\}$, we have $w \in A \cdot A^0 = A^1$. So $A \subset A^1$.

$A^1 \subset A$ and $A \subset A^1$; thus $A^1 = A$. □

Definition 3.18. If A is a language, then A^*, the *Kleene closure of* A [HMU06], denotes the set of words w such that $w \in A^n$ for some $n \in \mathbb{N}$.

We emphasize here that even though Σ, the set of letters, and Σ^1 the set of all single-letter words are defined separately, they are clearly isomorphic. It is common in the literature to abuse the notation and simply refer to letters as single-letter words— to claim $\Sigma^1 = \Sigma$ and consequently $\Sigma \subset \Sigma^*$ (Notation 3.8). However, when implementing in a programming language the distinction may be important in terms of programmatic types of the objects. Σ is a set of characters, while Σ^1 is a set of single character strings. The type of Σ and the type of Σ^1 are different, but the mapping between them is trivial.

Also note that $\varepsilon \in \Sigma^*$, $\varepsilon \notin \Sigma^1$, and $\Sigma^1 \subset \Sigma^*$. Some languages have finite cardinally, while others have countably infinite cardinally.

Definition 3.19. A *rational language* is defined by a recursive definition: The two sets $\varnothing$ and $\{\varepsilon\}$ are rational languages. For each letter of the alphabet, the singleton set containing the corresponding one letter word is rational language. In addition to these base definitions, any set which is the union or concatenation of two rational languages is a rational language. The Kleene closure of a rational language is a rational language.

Notation 3.20. Let $\mathcal{L}_\Sigma$ denote the set of all rational languages.

In light of Definition 3.19 and Notation 3.20, the definition of rational language can be restated by:

1. $\varnothing \in \mathcal{L}_\Sigma$.
2. $\{\varepsilon\} \in \mathcal{L}_\Sigma$.
3. $\{a\} \in \mathcal{L}_\Sigma \ \forall \ a \in \Sigma^1$.
4. $(A \cup B) \in \mathcal{L}_\Sigma \ \forall \ A, B \in \mathcal{L}_\Sigma$.
5. $(A \cdot B) \in \mathcal{L}_\Sigma \ \forall \ A, B \in \mathcal{L}_\Sigma$.
6. $A^* \in \mathcal{L}_\Sigma \ \forall \ A \in \mathcal{L}_\Sigma$.

While not part of the definition as such, it can be proven that if $A, B \in \mathcal{L}_\Sigma$ then $A \cap B \in \mathcal{L}_\Sigma$ and $A \backslash B \in \mathcal{L}_\Sigma$ [HMU06].

3.2 Rational expressions

The definition of rational language given in Section 3.1 provides a top-down mechanism for identifying regular languages. *I.e.*, languages are rational if they can be decomposed into other rational languages via certain set operations such as union, intersection, and concatenation. Conversely, new rational languages can be *discovered* by combining given rational languages in well defined ways.

Another way to identify rational languages is a bottom-up approach. This approach is based on the letters, rather than the sets. Rational expressions allow us to specify pattern based rules for determining which words are in a given language. We will say that a rational expression *generates* a language.

A rational expression is an algebraic expression, using the intuitive algebraic operators. A rational expression *generates* a language.

Notation 3.21. The notation $L = [\![r]\!]$, means that the rational expression, r generates the language L.

Notation 3.22. We denote the set of all rational expressions as $\mathcal{E}_{rat}$.

$$[\![\varnothing]\!] = \varnothing \qquad [\![\varepsilon]\!] = \{\varepsilon\} \qquad \forall a \in \Sigma^1, [\![a]\!] = \{a\}$$
$$[\![r + s]\!] = [\![r]\!] \cup [\![s]\!] \qquad [\![r^*]\!] = [\![r]\!]^* \qquad [\![rs]\!] = [\![r \cdot s]\!] = [\![r]\!] \cdot [\![s]\!] \qquad [\![r \cap s]\!] = [\![r]\!] \cap [\![s]\!]$$

This abuse of notation is commonplace in rational language theory. The same symbol a is used to denote a letter, $a \in \Sigma$, a word of length one, $a \in \Sigma^1$, and a rational expression, a *s.t.*$[\![a]\!] = \{a\} \subset \Sigma^*$. Analogously, the symbol ε is abused to denote both the empty word, $\varepsilon \in \Sigma^*$, and a rational expression ε *s.t.* $[\![\varepsilon]\!] = \{\varepsilon\} \subset \Sigma^*$. Further, $\varnothing \subset \Sigma^*$ denotes the empty language, and also the rational expression, $\varnothing$ *s.t.* $[\![\varnothing]\!] = \varnothing \subset \Sigma^*$.

We have proven (Theorems 3.11 and 3.13) that $\cdot$ is associative. It can also be proven that the operation $+$ is associative [HMU06], which means that without ambiguity we may write $(a{+}b{+}c)$ and $(a{\cdot}b{\cdot}c)$ omitting additional parentheses. However, we must define a precedence order to give an unambiguous meaning to expressions such as $a^*b{+}c{\cdot}d^*$. The precedence order from highest precedence to lowest is defined to be $(^*, \cdot, +)$, so that $a^*b{+}c{\cdot}d^*$ unambiguously means $((a^*) \cdot b) + (c \cdot (d^*))$.

As an example, let Σ be a language and $\{a, b, c, d, e, f\} \subset \Sigma$; the rational expression $a \cdot (b \cdot d^* + c \cdot e^*) \cdot f$, or equivalently $a(bd^* + ce^*)f$, can be understood to be a rational expression generating the set (language) of words which start with exactly one a, end with exactly one f, and between the a and f is either exactly one b followed by zero or more d's or exactly one c followed by zero or more e's.

The definition trivially implies that

$$\forall r \in \mathcal{E}_{rat}\ \exists R \in \mathcal{L}_\Sigma \mid [\![r]\!] = R \in \mathcal{L}_\Sigma\,,$$

and conversely,

$$\forall R \in \mathcal{L}_\Sigma \exists r \in \mathcal{E}_{rat} \mid [\![r]\!] = R\,.$$

3.3 Regular expressions

We would like to avoid confusion between the terms `regular expression` and `rational expression`. We use the term `regular expression` to denote programmatic implementations such as provided in `grep` and `Perl`. We assume the reader is familiar with UNIX-based regular expressions.

By contrast, we reserve the term *rational expression* to denote the algebraic expressions as described in Section 3.2.

There are regular expression libraries available for a wide variety of programming languages. Each implementation uses different ASCII characters to denote the rational language operations, often equipped with additional operations which are eventually reducible to the atomic operations shown above, and whose inclusion in the implementation adds expressivity in terms of syntactic sugar.

One of the oldest applications of regular expressions was in specifying the component of a compiler called a "lexical analyzer". The UNIX command `lex` allows the specification of tokens in terms of regular expressions in UNIX style and associates code to be executed when such a token is recognized [HMU06].

The same style regular expressions are built into several standard UNIX utilities such as `grep`, `egrep`, `sed` and several other programs. These implementations provide useful notations such as shown in Example 3.23.

Example 3.23 (Regular expression notation)**.**

symbol	regular expression	meaning
+	`"ab+c"`	one or more times, is equivalent to $a \cdot b \cdot b^* \cdot c$
?	`"ab?c"`	zero or one time, is equivalent to $a \cdot (b + \varepsilon) \cdot c$
.	`"a.c"`	any single character, is equivalent to $a \cdot \Sigma^1 \cdot c$

3.4 Finite automata

Finite automata provide a computational model for implementing recognizers for rational languages [ORT09b].

Definition 3.24. A *NDFA* (Non-Deterministic Finite Automaton) A is a 5-tuple $A = (\Sigma, Q, I, F, \delta)$ where:

Σ is an alphabet, (an alphabet is finite by definition)

Q is a finite set whose elements are called *states*

$I \subset Q$ is a set whose elements are called *initial states*

$F \subset Q$ is a set whose elements are called *final states*

$\delta \subset Q \times \Sigma \times Q$ is a set whose elements are called *transitions*.

Definition 3.25. In the special case that for each $(a, q_0) \in \Sigma \times Q$ there is at most one $q_1 \in Q$ such that $(q_0, a, q_1) \in \delta$, then the NDFA is referred to as a Deterministic Finite Automaton.

One might reasonably object to the terminology because despite the name, not all NDFAs are explicitly non-deterministic. Although the definitions are somewhat confusing, they are standard. Every DFA is an NDFA, but some NDFAs are not a DFA.

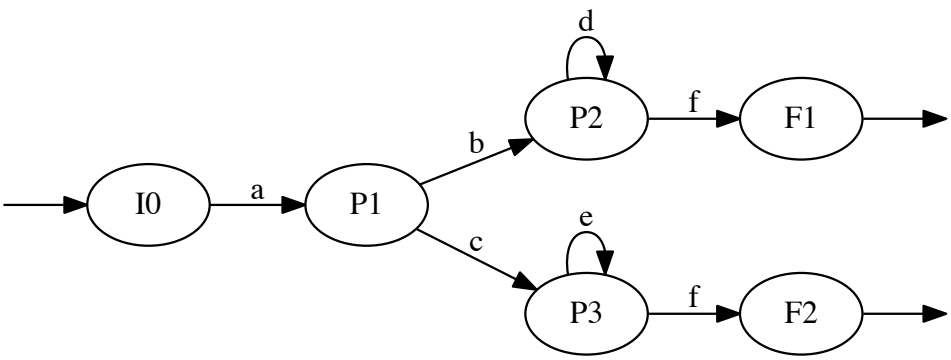


Figure 3.1: DFA recognizing the rational expression, $a \cdot (b \cdot d^* + c^+) \cdot f$

Each transition can be denoted $\alpha \xrightarrow{a} \beta$ for $\alpha, \beta \in Q$ and $a \in \Sigma$. Figure 3.1 shows a deterministic finite automaton. It has initial state $I = \{I_0\}$, final states $F = \{F_1, F_2\}$, and the following transitions:

$$\delta = \{I_0 \xrightarrow{a} P_1, P_1 \xrightarrow{b} P_2, P_2 \xrightarrow{d} P_2, P_2 \xrightarrow{f} F_1, P_1 \xrightarrow{c} P_3, P_3 \xrightarrow{e} P_3, P_3 \xrightarrow{f} F_1\,.$$

3.5 Equivalence of rational expressions and finite automata

It has been proven [HMU06] that the following statements are equivalent.

1. $L \in \mathcal{L}_\Sigma$
2. $\exists r \in \mathcal{E}_{rat} \mid L = [\![r]\!]$
3. $L \subset \Sigma^*$ is recognizable by a finite automaton

In fact, Figure 3.1 illustrates a finite automaton which recognizes the regular expression `a(bd*+ce*)f`.

3.6 The rational expression derivative

In 1964, Janusz Brzozowski [Brz64] introduced the concept of the *Rational Language Derivative*, and provided a theory for converting a regular expression to a DFA. Additional work was done by Scott Owens *et al.* [ORT09b] which presented the algorithm in easy to follow steps.

To understand what the rational expression derivative is and how to calculate it, first think of a rational expression in terms of its language, *i.e.* the set of sequences the expression *generates*. For example, the language of $((a|b) \cdot c^* \cdot d)$ is the set of words (finite sequences of letters) which begin with exactly one letter `a` or exactly one letter `b`, end with exactly one letter `d` and between contain zero or more occurrences of the letter `c`.

The *derivative of the language* with respect to a given letter is the set of suffixes of words which have the given letter as prefix. Analogously, the *derivative of the rational expression* is the rational expression which generates that language, *e.g.*, $\partial_a((a|b) \cdot c^* \cdot d) = (c^* \cdot d)$.

The Owens [ORT09b] paper explains a systematic algorithm for symbolically calculating such derivatives. The formulas listed in Figure 3.3 detail the calculations which must be recursively applied to calculate the derivative.

There are several algorithms for generating a finite automaton from a given rational expression. One very commonly used algorithm was inspired by Ken Thompson [YD14, Xin04] and involves straightforward pattern substitution. While this algorithm is easy to implement it has a serious limitation. It is not able to easily express automata resulting from the intersection of two rational expressions.

Because of this limitation, we have chosen to use the algorithm based on regular expression *derivatives*. This algorithm was first presented in 1964 by Janusz Brzozowski [Brz64]. While Brzozowski's result was applied to digital circuits, Scott Owens *et al.* [ORT09b] extended the principle to generalize regular pattern recognition for sequences of characters.

We will define the derivate later, but we find it useful to first define nullability. As will be seen below, to calculate the derivative of some rational expressions, we must calculate whether the rational expression is nullable.

Definition 3.26. A rational language is said to be *nullable* if it contains the empty word; *i.e.*, a language $L \subset \Sigma^*$ is *nullable* if $\varepsilon \in L$. Likewise, a rational expression r is *nullable* if $[\![r]\!]$ nullable.

The function ν (the Greek letter nu) calculates nullability.

$$\nu : \mathcal{E}_{rat} \to \{\varnothing, \varepsilon\} \subset \Sigma^*$$

as defined according to the recursive rules in Figure 3.2. If $\nu(r) = \varepsilon$ then r is nullable. If $\nu(r) = \varnothing$ then r is not nullable.

It may be worth reëmphasizing here that the symbol ε in Definition 3.26 is the rational expression ε not the empty word. As mentioned in Section 3.2 this abuse of notation is commonplace in the literature pertaining to rational language theory. This notational abuse comes up again in the proof of Lemma 3.31.

$$\nu(\varnothing) = \varnothing \tag{3.1}$$
$$\nu(\varepsilon) = \varepsilon \tag{3.2}$$
$$\nu(a) = \varnothing \ \forall\ a \in \Sigma, a \neq \varepsilon \tag{3.3}$$
$$\nu(r + s) = \nu(r) + \nu(s) \tag{3.4}$$
$$\nu(r \cdot s) = \nu(r)\ \cap\ \nu(s) \tag{3.5}$$
$$\nu(r\ \cap\ s) = \nu(r)\ \cap\ \nu(s) \tag{3.6}$$
$$\nu(r^*) = \varepsilon \tag{3.7}$$

Figure 3.2: Recursive rules defining the nullability function ν

Definition 3.27. Given a language $L \subset \Sigma^*$ and a word $w \in \Sigma^*$, the *derivative of L with respect to w* denoted $\partial_w L$ is a language

$$\partial_w L = \{v \mid w \cdot v \in L\}.$$

If $[\![S]\!] = L$, and $w \in L$, then a *derivative of S with respect to w* is denoted $\partial_w S$. Moreover, $\partial_w S \in \mathcal{E}_{rat}$ and $[\![\partial_w S]\!] = \partial_w L$. Otherwise stated, we can speak of either the derivative of the language L or the derivative of a rational expression.

Example 3.28 (Rational derivative)**.** For example, suppose $L = \{this, that, those, fred\}$, then $\partial_{th} L = \{is, at, ose\}$. Basically take the words which start with the given prefix, and remove the prefix.

It can be proven that if $L \in \mathcal{L}_\Sigma$, then $\partial_w L \in \mathcal{L}_\Sigma\ \forall\ w \in \Sigma^*$ [ORT09b]. However, it is not implied nor is it true in general that $\partial_w L \subset L$.

Given a rational expression, we would like to calculate the rational expression representing its derivative. To do this, the reduction rules shown in Figure 3.3 can be recursively applied.

$$\partial_a \varnothing = \varnothing \tag{3.8}$$
$$\partial_a \varepsilon = \varnothing \tag{3.9}$$
$$\partial_a a = \varepsilon \tag{3.10}$$
$$\partial_a b = \varnothing \text{ for } b \neq a \tag{3.11}$$
$$\partial_a(r + s) = \partial_a r + \partial_a s \tag{3.12}$$
$$\partial_a(r \cdot s) = \begin{cases} \partial_a r \cdot s, & \text{if } \nu(r) = \varnothing \quad (3.13) \\ \partial_a r \cdot s + \partial_a s, & \text{if } \nu(r) = \varepsilon \quad (3.14) \\ \partial_a r \cdot s + \nu(r) \cdot \partial_a s, & \text{in either case} \quad (3.15) \end{cases}$$
$$\partial_a(r\ \cap\ s) = \partial_a r\ \cap\ \partial_a s \tag{3.16}$$
$$\partial_a(r^*) = \partial_a r \cdot r^* \tag{3.17}$$
$$\partial_\varepsilon r = r \tag{3.18}$$
$$\partial_{u \cdot v} r = \partial_v(\partial_u r) \tag{3.19}$$

Figure 3.3: Rules for the Brzozowski derivative

Note that (3.15) is useful for theoretical and hand calculation but is problematic for algorithmic calculation. In the case that $\nu(r)$ is $\varnothing$, (3.15) is equivalent to (3.13), but special attention is required because a naïve implementation of $\partial_a s$ may result in an infinite recursion. To avoid such an error, (3.13) (3.14) should be used instead of (3.15).

Notation 3.29. For any rational expression r, we use the notation r^+ to mean exactly $r \cdot r^*$

In Section 4.4.3 we will need expressions for $\partial_a r^+$ and $\nu(r^+)$, so we derive them now.

Theorem 3.30. *r^+ and r^* have the same derivative.*

$$\partial_a r^+ = \partial_a r^*$$

Proof. By Notation 3.29 we have,

$$\partial_a r^+ = \partial_a (r \cdot r^*) \,.$$

There are two cases possible: either $\nu(r) = \varnothing$ or $\nu(r) = \varepsilon$.

If $\nu(r) = \varnothing$:

$$\partial_a (r \cdot r^*) = (\partial_a r) \cdot r^* \text{by (3.13)}$$

If $\nu(r) = \varepsilon$:

$$\begin{aligned} \partial_a (r \cdot r^*) &= (\partial_a r) \cdot r^* + \partial_a r^* && \text{by (3.14)} \\ &= (\partial_a r) \cdot r^* + \partial_a r \cdot r^* && \text{by (3.17)} \\ &= (\partial_a r) \cdot r^* \end{aligned}$$

So in either case

$$\partial_a (r \cdot r^*) = (\partial_a r) \cdot r^* = \partial_a r^* \,.$$

□

Lemma 3.31.

$$\nu(r) \cap \varepsilon = \nu(r)$$

Proof. There are two cases possible $\nu(r) \in \{\varnothing, \varepsilon\}$. If $\nu(r) = \varnothing$, then $\nu(r) \cap \varepsilon = \varnothing \cap \varnothing = \varnothing = \nu(r)$. If $\nu(r) = \varepsilon$, then $\nu(r) \cap \varepsilon = \varepsilon \cap \varepsilon = \varepsilon = \nu(r)$. In either case $\nu(r) \cap \varepsilon = \nu(r)$. □

In the proof of Lemma 3.31 we refer to expressions such as $\varepsilon \cap \varepsilon$, by which we mean $\{\varepsilon\} \cap \{\varepsilon\}$, ε in the latter case being the empty word $\varepsilon \in \Sigma^*$.

Theorem 3.32. *r^+ is nullable if and only if r is nullable.*

$$\nu(r^+) = \nu(r)$$

Proof.

$$\begin{aligned} \nu(r^+) &= \nu(r \cdot r^*) && \text{By Notation 3.29} \\ &= \nu(r) \cap \nu(r^*) && \text{By (3.5)} \\ &= \nu(r) \cap \varepsilon && \text{By (3.7)} \\ &= \nu(r) && \text{By Lemma 3.31} \end{aligned}$$

□

3.7 Computing the automaton using the rational derivative

To compute the automaton corresponding to a rational expression [ORT09b], use Algorithm 1. The algorithm assumes the existence of constructor functions `State` (which creates a state given a rational expression), `Transition` (which creates a transition given two states, the origin and destination, and an element of the alphabet to serve as label), and `Automaton` (which creates an automaton as described in Definition 3.25).

Brzozowski argued that this procedure terminates because there is only a finite number of derivatives possible, modulo multiple equivalent algebraic forms. Eventually all the expressions encountered will be algebraically equivalent to the derivative of some other expression in the set.

Algorithm 1: Computes an automaton using the rational derivative

Input: r : a rational expression
Input: Σ : alphabet
Output: An automaton as defined in Definition 3.25

```
1.1  begin
1.2      F ← ∅                                                  // set of final states
1.3      q ← State(r)
1.4      if ν(r) then
1.5          F ← F ∪ {q}
1.6      I ← {q}                                                // set of initial states
1.7      Q ← {q}                                                // set of states
1.8      W ← {q}                                                // set of TODO work states
1.9      while W ≠ ∅ do
1.10         q_0 ← any element from W
1.11         W ← W\{q_0}
                                  // Iterate over all the possible 1-letter words
1.12         for a ∈ Σ^1 do
1.13             r ← q_0.expression                  // the expression associated with the state
1.14             d ← ∂_a r
1.15             Reduce d to canonical form
1.16             if d = ∅ then
1.17                 Nothing
1.18             else if ∃q ∈ Q such that q.expression = d then
1.19                 δ ← Transition(q_0, q, a)
1.20             else
1.21                 q ← State(d)
1.22                 δ ← Transition(q_0, a, q)
1.23                 W ← W ∪ {q}
1.24                 Q ← Q ∪ {q}
1.25                 if ν(d) then
1.26                     F ← F ∪ {q}
1.27     return Automaton(Σ,Q,I,F,δ)
```

There are a couple of useful optimization steps.

If the derivative is $\varnothing$, there is really no reason to explicitly add the a null state to the automaton. Doing so would clutter the graphical representation with arrows leading to this state.

It is not necessary that there will be 1:1 correspondence between the non-trivial derivatives and the states. The problem is that reducing the rational expressions to a canonical form is a hard problem, since many rational expressions may generate the same rational language. Even so, one would expect that there might be one *canonical reduced* expression which could be arrived at, given a finite set of identities such as $\varnothing + L = L = L + \varnothing, \varepsilon \cdot L = L = L \cdot \varepsilon, (L^*)^* = L^*, L + K = K + L$, *etc.* In fact, there is no finite set of identities which permits to deduce all identities between rational expressions [Pin15].

It suffices to allow the same derivative in two different algebraic forms to be represented by multiple states as long as it is a reasonable number. There must be some reduction step in the derivative calculation to limit the number of forms expressed, but the reduction need not actually reduce every expression to a unique, canonical form.

3.8 Related work

There are many sources of information on rational language theory, a few are [HMU06, YD14].

The PCRE (Perl Compatible Regular Expressions) [Haz] library available in many languages such as C, SKILL [Bar90] represent the rational expression shown above as `"a(bd*|cd*)f"`. An implementation of PCRE for Common Lisp is available and in wide usage, CL-PPCRE [Wei15].

Even though we chose the algorithm based on Brzozowski derivatives, there is another commonly used algorithm, presented by Yvon and Demaille [YD14] for constructing a DFA which was inspired by Ken Thompson [YD14, Xin04]. This alternate algorithm involves decomposing a rational expression into a small number of cases such as base variable, concatenation, disjunction, and Kleene star, then following a graphical template substitution for each case. While this algorithm is easy to implement, it has a serious limitation. It is unable to easily express automata resulting from the intersection or complemention of rational expressions. We rejected this approach as we would like to support regular type expressions (Section 4.2.1) containing the keywords `:and` and `:not`, such as in `(:and (:* t integer) (:not (:* float t)))`.

Chapter 4

Type-Checking of Heterogeneous Sequences in Common Lisp

In this chapter we extend the work from Chapters 2 and 3. In Chapter 2 we introduced types in the Common Lisp language, primarily concentrating on atomic objects. In Chapter 3 we summarized the theory of rational languages, including an algorithm to construct a finite state machine which recognizes sequences of characters in a given rational language.

In this chapter, we extend both theories to accommodate regular sequences of Common Lisp types. We do so by extending the Common Lisp type system to accommodate rational type expressions. These expressions allow us to express patterns of types within heterogeneous sequences. We present the Common Lisp implementation of regular type expressions including some analysis of their performance against other reasonable approaches.

The theoretical work we present in this chapter is sound, although an initial implementation reveals several challenges of representation and efficient execution. Correct conversion of a rational type expression into a deterministic finite state machine requires, first, decomposing a set of types into an equivalent partition of types (the MDTD, maximal disjoint type decomposing, problem), and second, serializing this decomposition into Common Lisp code. In Chapters 8 and 9 we describe algorithms and performance analysis of the MDTD problem. In Chapter 11 we describe solutions to the serialization problem.

Rather than jumping directly from Chapter 4 to Chapter 8 we first develop a computation tool, the Binary Decision Diagram, which we use both in the MDTD problem and also the serialization problem. We introduce this data structure in Chapter 5, examine many of its properties in Chapter 6, and extend it to accommodate Common Lisp types in Chapter 7 before proceeding on to attacking the MDTD problem.

4.1 Introduction

As explained in Section 2.1, a *type* in Common Lisp [Ans94] is identically a set of (potential) values at a particular point in time during the execution of a program [Ans94, Section 4.1]. Information about types provides clues for the compiler to make optimizations such as for performance, space (image size), safety or debuggability [New15, Section 4.3] [Ans94, Section 3.3]. Application programmers may as well make explicit use of types within their programs, such as with `typecase`, `typep`, `the`, `type-of`, `check-type`, *etc.*

Using the existing Common Lisp type system, the programmer can specify a homogeneous type for all the elements of a vector [Ans94, Section 15.1.2.2], or the type for a particular element of a list [Ans94, System Class CONS]. Two notable limitations which we address in this report are: 1) that there is no standard way to specify heterogeneous types for different elements of a vector; 2) neither is there a standard way to declare types (whether heterogeneous or homogeneous) for all the elements of a list. See Section 4.5 for a vain attempt.

The small code snippet in Example 4.1 shows the Common Lisp notation for allocating a one dimensional array of 10 elements, each of which are integer, and a one dimensional array of 256 elements each of which may be either a string or a number.

Example 4.1 (Creating a one dimensional array object with declared element type)**.**

```
(make-array '(10) :element-type 'integer)
(make-array '(256) :element-type '(or string number))
```

We introduce the concept of *rational type expression* for abstractly describing patterns of types within sequences. The concept is envisioned to be intuitive to the programmer in that it is analogous to patterns described by regular expressions, which we assume the reader is already familiar with.

Just as the characters of a `string` may be described by a rational expression such as $(a \cdot b^* \cdot c)$, which intends to match strings such as `"ac"`, `"abc"`, and `"abbbbc"`, the rational type expression $(string \cdot number^* \cdot symbol)$ is intended to match vectors like `#("hello" 1 2 3 world)` and lists like `("hello" world)`. Rational expressions match character constituents of strings according to character equality. Rational type expressions match elements of sequences by element type.

We further introduce an s-expression based syntax, called *regular type expression* to encode a *rational type expression*. This syntax replaces the infix and postfix operators in the rational type expression with prefix notation based s-expressions. The regular type expression `(:cat string (:* number) symbol)` corresponds to the rational type expression $(string \cdot number^* \cdot symbol)$. In addition, we provide a parameterized type named `rte`, whose argument is a regular type expression. The members of such a type are all sequences matching the given regular type expression. Section 4.2.1 describes the syntax.

As the Lisp programmer would expect, the `rte` type may be used anywhere within a Lisp program where a type specifier is expected, as suggested in Example 4.5. See section 4.3.4 for more details of list destructuring. See Section 4.3 for several examples.

While we avoid making claims about the potential utility of declarations of such a type from the compiler's perspective [com15], there would be numerous and obvious challenges posed by such an attempt. At run-time, `cons` cells may be freely altered because Common Lisp does not provide read-only `cons` cells. There is a large number of Common Lisp functions which are allowed to modify `cons` cells, thus violating the proposed type constraints if left unchecked. Additionally, if declarations were made about certain lists, and thereafter other lists are created (or modified) to share those tails, it is not clear which information about the tails should be maintained.

Nevertheless, we do suggest that a declarative system to describe patterns of types within sequences (vectors and lists) would have great utility for program logic, code readability, and type safety.

A discussion of the theory of rational languages on which our research is grounded, may be found in [HMU06, Chapters 3,4].

4.2 Heterogeneous sequences in Common Lisp

The Common Lisp language supports heterogeneous sequences in the form of sequentially accessible lists and several arbitrarily accessible vectors. A sequence is an ordered collection of elements, implemented as either a vector or a list [Ans94]. The Lisp reader recognizes syntax supporting several types of sequence. Several are shown in Example 4.2.

Example 4.2 (Some Common Lisp sequences)**.**

```
"a string is a sequence of characters"
(list of 9 elements including "symbols" "strings and" a number)
#(vector of 9 elements including "symbols" "strings and" a number)
```

4.2.1 The regular type expression

We have implemented a parameterized type named `rte` (regular type expression), via `deftype`. The argument of `rte` is a *regular type expression*. Some specifics of the implementation are explained in Section 4.4.

Definition 4.3. A *regular type expression* is defined as either a Common Lisp type specifier, such as `number`, `(cons number)`, `(eql 12)`, or `(and integer (satisfies oddp))`, or a list whose first element is one of a limited set of keywords shown in Figure 4.2, and whose trailing elements are other regular type expressions.

The grammar of a regular type expressions is given in Figure 4.1.

$$\langle \text{regular-type-expression} \rangle \models \langle \text{cl-type-specifier} \rangle \mid \langle \text{literal} \rangle \mid \langle \text{compound} \rangle \quad (4.1)$$
$$\langle \text{literal} \rangle \models \text{:empty-word} \mid \text{:empty-set} \quad (4.2)$$
$$\langle \text{compound} \rangle \models \langle \text{compound-1-arg} \rangle \mid \langle \text{compound-var-args} \rangle \quad (4.3)$$
$$\langle \text{compound-1-arg} \rangle \models (\langle \text{1-arg-op} \rangle \langle \text{regular-type-expression} \rangle) \quad (4.4)$$
$$\langle \text{compound-var-args} \rangle \models (\langle \text{1-arg-op} \rangle \langle \text{exprs} \rangle) \quad (4.5)$$
$$\langle \text{var-args-op} \rangle \models \text{:cat} \mid \text{:and} \mid \text{:or} \quad (4.6)$$
$$\langle \text{1-arg-op} \rangle \models \text{:*} \mid \text{:+} \mid \text{:?} \mid \text{:not} \quad (4.7)$$
$$\langle \text{exprs} \rangle \models \langle \text{rational-type-expression} \rangle \mid \langle \text{rational-type-expression} \rangle \langle \text{exprs} \rangle \quad (4.8)$$

Figure 4.1: Regular type expression grammar

Example 4.4 (Not a valid `rte` type specifier)**.** As a counter example, `(rte (:cat (number number)))` is invalid because `(number number)` is neither a valid Common Lisp type specifier, nor does it begin with a keyword from Figure 4.2.

There is an additional syntax not explained in the grammar in Figure 4.1. Normally the `:*`, `:+` and `:?` regular type expression keywords take a single argument such as `(rte (:* number))`. However, if they are used with multiple arguments such as `(rte (:* symbol number))`, `(rte (:+ p symbol number))`, or `(rte (:? symbol number))`, they are interpreted internally with an implicit `:cat` inserted, such as `(rte (:* (:cat symbol number)))`, `(rte (:+ (:cat symbol number)))`, and `(rte (:? (:cat symbol number)))`.

Example 4.5 (Declaring a function which a declared argument of type `plist`)**.**

```
(assert (typep my-list '(rte (:cat mytype number))))

(deftype plist ()
  `(rte (:* symbol t)))

(defun F (object plist list-of-int)
  (declare (type plist plist)
           (type (and list (rte (:* integer))) list-of-int))
  (typecase object
    ((rte (:cat symbol (:* number)))
     (destructuring-bind (name &rest numbers) object
        ...))
    ((rte (:cat symbol list (:* string)))
     (destructuring-bind (name data &rest strings) object
        ...))))
```

Example 4.6 declares a class whose `point` slot is a list of two numbers. A subtlety to note is that `rte` is a subtype of `sequence` not of `list`. This means that the type `(rte (:cat number number))` includes not only the list `(1 2.0)` but also the vector `#(1 2.0)`.

Example 4.6 (Declaring a class with `point` slot declared as an `rte` type)**.**

```
(defclass F ()
  ((point :type (and list (rte (:cat number number))))
   #|...|#))
```

Keyword	Description	Example
`:*`	Match zero or more times. The example matches a sequence of zero or more objects of type `string`. *e.g.*, `()`, `("abc")`, or `("abc" "xyz")`, but not `("abc" "xy" 1.2)`.	`(:* string)`
`:cat`	Concatenate zero or more regular type expressions. The example matches a sequence of `number` followed by `string` followed by `list`.	`(:cat number string list)`
`:empty-set`	Does not match any words. `:empty-set` is the identity element for `:or`, and is useful for internal representations. Not really useful for the end user.	
`:empty-word`	The empty word is a standard concept in rational languages. It is useful in combinations with `:or` and `:cat`. The example matches a sequence of one or two strings.	`(:cat string` `  (:or string :empty-word))`
`:or`	Match any of the regular type expressions. The example matches a sequence which consists either of all strings or all symbols.	`(:or (:* string) (:* symbol))`
`:and`	Match all of the regular type expressions. The example matches a sequence which starts with two strings, and also ends with two strings.	`(:and (:cat string string (:* t))` `  (:cat (:* t) string string))`
`:not`	Match a sequence which does NOT match the given regular type expressions. The example matches any sequence except one which is one or more repetition of `keyword string`.	`(:not (:+ (:cat keyword string)))`
`:+`	Match one or more times. The example matches `("1" 2 (3))` but not the empty list.	`(:+ string number list)`
`:?`	Match zero or one time. The example matches `()` and `("abc" 1.2 (a b c))`, but not `("abc" 1.2 (a b c) "xyz" 3 ())`.	`(:? (:cat string number list))`

Figure 4.2: Regular type expression keywords

Example 4.7 is the definition of a function whose second argument must be a list of exactly 2 strings or 3 numbers.

Example 4.7 (Declaration of function whose second argument is declared via `rte`).

```
(defun F (X Y)
  (declare (type Y (and list
                         (rte (:or (:cat number number number)
                                   (:cat string string))))))
  #|...|#)
```

In Example 4.8 we declare types named `point-2d`, `point-3d`, and `point-sequence` which can be used in other declarations:

Example 4.8 (Definitions of `point-2d` and `point-3d`).

```
(deftype point-2d ()
  "A list of exactly two numbers."
  '(and list (rte (:cat number number))))

(deftype point-3d ()
  "A list of exactly three numbers."
  `(and list (rte (:cat number number number))))

(deftype point-sequence ()
  "A list or vector of points, each point may be 2d or 3d."
  '(rte (:or (:* point-2d) (:* point-3d))))
```

Example 4.9 (Examples of syntax using `rte`).

`(rte (:cat number number number))`
corresponds to the rational type expression $(number \cdot number \cdot number)$ and matches a sequence of exactly three numbers.

`(rte (:or number (:cat number number number)))`
corresponds to $(number + (number \cdot number \cdot number))$ and matches a sequence of either one or three numbers.

`(rte (:cat number (:?  number number)))`
corresponds to $(number \cdot (number \cdot number)^{?})$ and matches a sequence of one mandatory number followed by exactly zero or two numbers. This happens to be equivalent to the previous example.

`(rte (:* cons number))`
corresponds to $(cons \cdot number)^{*}$ and matches a sequence of a `cons` followed by a `number` repeated zero or more times, *i.e.*, a sequence of even length.

4.2.2 Clarifying some confusing points about regular type expressions

There are a couple of potentially confusing points to note about the syntax of the regular type expression.

The argument of `rte` is a regular type expressions, which may be either Common Lisp specifier or another regular type expression. Consider Example 4.10 with the `cons` type specifier. In Common Lisp an object of type `cons` is a non-`nil` list. An object of type `(cons number)` is a list whose first element is of type `number`.

Example 4.10 (rte using `cons`)**.**

`(rte (:cat cons number))` — A sequence of length 2 whose respective elements are a non-empty list and a number.

`(rte (cons number))` — A sequence of length 1 whose element is a list whose first element is a number.

`(rte (:cat (cons number)))` — Same as `(rte (cons number))`.

Another potentially confusing point about the syntax is that `and` and `:and` (similarly `or` and `:or`) may both be used but have different meanings in most cases. The Common Lisp type specifiers, `and` and `or` match exactly one object. For example: `(or string symbol)` matches one object which must either be a `string` or a `symbol`. The operands of `and` and `or` are Common Lisp type specifiers. For example `(and (:* string) (:* number))` is not valid because `(:+ string)` and `(:* number)` are not valid Common Lisp type specifiers.

Contrast that with the regular type expression keywords `:and` and `:or` whose operands are regular type expressions, which may happen to be type specifiers. For example `(rte (:or (:+ string) (:* number)))` is valid, and so is `(:or string number)`.

Additionally, regular type expressions may reference Common Lisp type specifiers. For example: `(rte (:or (:+ string) (and list (not null))))`, which matches either a non-empty sequence of strings, or a singleton sequence whose element is a non-empty list.

To avoid confusion, we emphasize the difference between `(:cat number symbol)` and `(rte (:cat number symbol))`. We refer to a form such as `(:cat number symbol)` as a regular type expression, and the corresponding Common Lisp type is specified by `(rte (:cat number symbol))`. In fact `(rte (:cat number symbol))` can be used, within Common Lisp code, anywhere a Lisp type specifier is expected. However, `(:cat number symbol)` is not a Common Lisp type specifier; it may only be used as a parameter to the `rte` type.

A subtle point worth repeating is that any Common Lisp type specifier is a valid regular type expression (but not vice versa). So `(rte (:cat number symbol))` may also be used where a regular type expression is expected, including being used recursively within another regular type expression. Compare the expressions in Example 4.11

Example 4.11 (Using `rte` recursively)**.**

`(rte (:cat number (rte (:cat symbol symbol))))` — matches a sequence of length exactly two, whose first element is a number, and whose second element is a sequence of exactly two symbols. E.g., `(1.1 (a b))`

`(rte (:cat number (:cat symbol symbol)))` — matches a sequence of length exactly three whose first element is a number, and whose next two elements are symbols. E.g., `(1.1 a b)`

4.3 Application use cases

The following subsections illustrate some motivating examples of regular type expressions.

4.3.1 Use Case: RTE-based string regular expressions

Since a string in Common Lisp is a sequence, the `rte` type may be used to perform simple string regular expression matching. Our tests have shown that the `rte` based string regular expression matching is about 35% faster than CL-PPCRE [Wei15] when restricted to features strictly supported by the theory of rational languages, thus ignoring CL-PPCRE features such as character encoding, capture buffers, recursive patterns, *etc.*

The call to the function `remove-if-not` in Example 4.12, filters a given list of strings, retaining only those that match an implicit regular expression `"a*Z*b*"`. The function, `regexp-to-rte` converts a string regular expression to a regular type expression.

Example 4.12 (Expansion of `rte` representing conventional string regular expression)**.**

```
(regexp-to-rte "(ab)*Z*(ab)*")
==>
  (:cat (:* (member #\a #\b))
      (:* (eql #\Z))
      (:* (member #\a #\b)))

(remove-if-not
  (lambda (str)
    (typep str
      `(rte ,(regexp-to-rte "(ab)*Z*(ab)*"))))
  '("baZab"
    "ZaZabZbb"
    "aaZbbbb"
    "aaZZZZbbbb"))
==>
("baZab"
 "aaZbbbb"
 "aaZZZZbbbb")
```

The `regexp-to-rte` function does not attempt the daunting task of fully implementing Perl compatible regular expressions as provided in CL-PPCRE. Instead `regexp-to-rte` implements a small but powerful subset of CL-PPCRE whose grammar is provided by [Cam99]. Starting with this context free grammar, we use `CL-Yacc` [Chr09] to parse a string regular expression and convert it to a regular type expression.

4.3.2 Use Case: Test cases based on extensible sequences

Climb [LSCCDV12] is an image processing library implemented in Common Lisp. It represents digital images in a variety of internal formats, including as a two dimensional array of pixels, or what conceptually serves the function of a 2D array. The image may be populated with pixel values such as RGB objects or gray-scale scalars, but may also have image-boundary (place-holder) elements which are not required to be valid pixel values. Certain image processing functions are expected to calculate new images. For testing purposes we would like to make assertions about rows and columns of the 2D arrays; *e.g.*, we'd like to assert that the row vectors and column vectors of a given image (excluding the border elements) are indeed RGB (red-green-blue) values, and that the row and column vectors in the calculated image are gray-scale values. Unfortunately, Common Lisp 2D arrays are not of type `sequence`, so 2D image arrays in the Climb platform, are not natively compatible with regular type expression based matchers.

To solve this problem, we exploit a feature of SBCL called *Extensible Sequences* [New15, Rho09]. The extensible sequence protocol requires an application like Climb, to implement CLOS [Ans94] methods on generic functions such as `length`, `elt`, and `(setf elt)` specializing on application specific classes such as `row-vector` and `column-vector` shown in Implementation 4.13. The application specific classes in this case, `row-vector` and `column-vector`, allow vertical and horizontal *slices* a 2D array to be viewed as a `sequence`.

Implementation 4.13.

```
(defclass 2d-array-sequence (sequence
                             standard-object)
  ((2d-array :initarg :2d-array
             :reader 2d-array)))

(defclass row-vector (2d-array-sequence)
  ((row :type fixnum
        :initarg :row
        :accessor row)))

(defclass column-vector (2d-array-sequence)
```

```
  ((column :type fixnum
           :initarg :column
           :accessor column)))

(defmethod sequence:length
     ((seq column-vector))
  (array-dimension (2d-array seq) 0))

(defmethod sequence:elt ((seq column-vector)
                         row)
  (aref (2d-array seq) row (column seq)))

(defmethod (setf sequence:elt)
      (value
       (seq column-vector)
       row)
  (setf (aref (2d-array seq) row (column seq))
        value))
```

The unit tests for Climb are implementing using `Lisp-Unit` [Rie]. The code in Example 4.14 shows such a test which loads an RGB image named `"lena128.bmp"` and makes some assertions about the format of the internal Lisp data structures. In particular it views the image as a sequence of row-vectors, the first and last of which may contain any content `(rte (:* t))`, but the rows in between are of the form `(rte t (:* rgb) t)`.

Example 4.14 (Unit test asserting types for row vectors)**.**

```
(define-test io/2d-array-b
  (let* ((rgb-image (image-load "lena128.bmp")))
         (seq (make-vector-of-rows rgb-image)))

    (assert-true
       (typep seq
         '(rte sequence
               (:* (rte t (:* rgb) t))
               sequence)))))
```

4.3.3 Use Case: DWIM lambda lists

In this section we look briefly at Do-What-I-Mean (DWIM) lambda lists. As a complex yet realistic example we use a regular type expression to test the validity of a Common Lisp lambda list, which are sequences which indeed are described by a pattern.

```
lambda-list := (var*
                [&optional {var | (var [init-form [supplied-p-parameter]])}*]
                [&rest var]
                [&key {var | ({var | (keyword-name var)}
                           [init-form [supplied-p-parameter]]) }*
                      [&allow-other-keys]]
                [&aux {var | (var [init-form])}*]
                )
```

Figure 4.3: CLHS Syntax of ordinary lambda list

Common Lisp specifies several different kinds of lambda lists, used for different purposes in the language. *E.g.*., the *ordinary lambda list* is used to define lambda functions, the *macro lambda list* is for defining macros, and the *destructuring lambda list* is for use with `destructuring-bind`. Each of these lambda lists has its own

syntax, the simplest of which is the *ordinary lambda list* (Figure 4.3). The code in Example 4.15 shows examples of ordinary lambda lists which obey the specification but may not mean what you think.

Example 4.15 (Functions with dubious lambda lists)**.**

```
(defun F1 (a b &key x &rest other-args)
  ...)

(defun F2 (a b &key ((Z U) nil u-used-p))
  ...)
```

The function `F1`, according to careful reading of the Common Lisp specification, is a function with three keyword arguments, `x`, `&rest`, and `other-args`, which can be referenced at the call site with a bizarre function calling syntax such as `(F1 1 2 :x 3 :&rest 4 :other-args 5)`. What the programmer probably meant was one keyword argument named `x` and an `&rest` argument named `other-args`. According to the Common Lisp specification [Ans94, Section 3.4.1], in order for `&rest` to have its normal *rest-args* semantics in conjunction with `&key`, it must appear before, not after, the `&key` lambda list keyword. The specification makes no provision for `&rest` following `&key` other than that one name a function parameter and the other have special semantics. This issue is subtle. In fact, SBCL considers this such a bizarre situation that it diverges from the specification and flags a `SB-INT:SIMPLE-PROGRAM-ERROR` during compilation: `misplaced &REST in lambda list: (A B &KEY X &REST OTHER-ARGS)`

The function `F2` is defined with an *unconventional* `&key` parameter which is not a symbol in the `keyword` package but rather in the current package. Thus the parameter `U` must be referenced at the call-site as `(F2 1 2 'Z 3)` rather than `(F2 1 2 :Z 3)`.

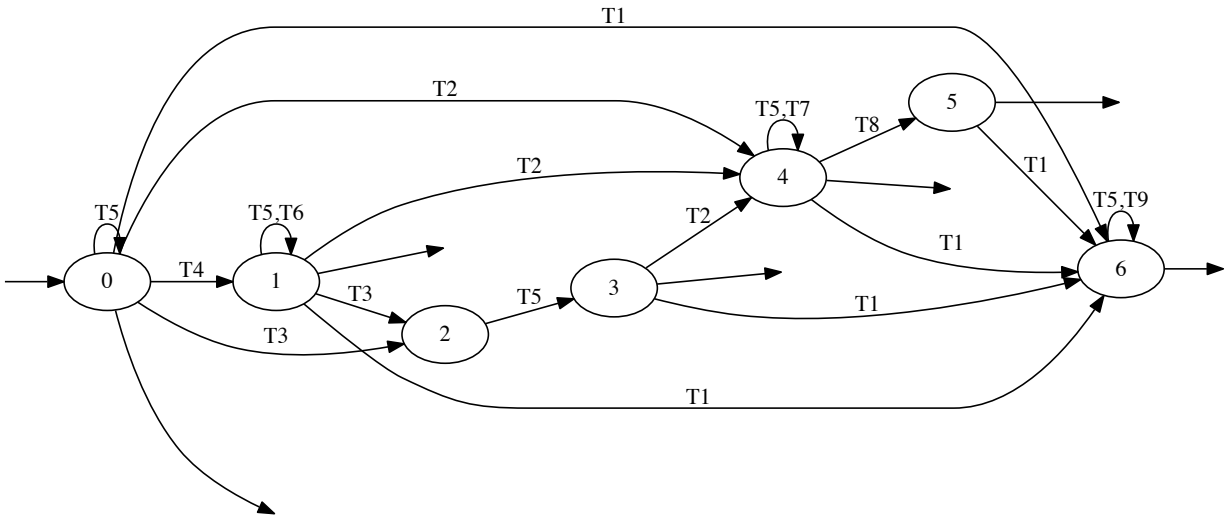
Implementation 4.16 (`deftype var`)**.**

```
(deftype var ()
  `(and symbol
        (not (or keyword
                 (member t nil)
                 (member ,@lambda-list-keywords)))))
```

Implementation 4.17 (`dwim-ordinary-lambda-list`)**.**

```
(deftype dwim-ordinary-lambda-list ()
  (let* ((optional-var '(:or var (:and list (rte (:cat var
                                                       (:? t
                                                           (:? var)))))))
         (optional `(:cat (eql &optional) (:* ,optional-var)))
         (rest '(:cat (eql &rest) var))
         (key-var '(:or var
                        (:and list
                              (rte (:or var (cons keyword
                                                  (cons var null)))
                                   (:? t
                                       (:? var))))))
         (key `(:cat (eql &key)
                     (:* ,key-var)
                     (:? (eql &allow-other-keys))))
         (aux-var '(:or var (:and list (rte (:cat var (:? t))))))
         (aux `(:cat (eql &aux) (:* ,aux-var))))

    `(rte
```

```
(:* var)
(:? ,optional)
(:? ,rest)
(:? ,key)
(:? ,aux))))
```



Transition label	Regular type expression
T_1	`(eql &aux)`
T_2	`(eql &key)`
T_3	`(eql &rest)`
T_4	`(eql &optional)`
T_5	`var`
T_6	`(and list (rte var (:? t (:? var))))`
T_7	`(and list (rte (:or var (cons keyword (cons var null))) (:? t (:? var))))`
T_8	`(eql &allow-other-keys)`
T_9	`(and list (rte var (:? t)))`

Figure 4.4: DFA recognizing conventional ordinary lambda list

These situations are potentially confusing, so we define what we call the *dwim ordinary lambda list*. Implementation 4.17 shows an implementation of the type `dwim-ordinary-lambda-list`. A Common Lisp programmer might want to use this type as part of a code-walker based checker. Elements of this type are lists which are indeed valid lambda lists for `defun`, even though the Common Lisp specification allows a more relaxed syntax. Figure 4.4 showing the corresponding DFA gives a vague idea of the complexity of the matching algorithm.

The *dwim ordinary lambda list* differs from the *ordinary lambda list*, in the aspects described above and also that it ignores semantics the particular Lisp implementation may have given to additional lambda list keywords. It only supports semantics for: `&optional`, `&rest`, `&key`, `&allow-other-keys`, and `&aux`.

4.3.4 Use Case: `destructuring-case`

Example 4.18 (`typecase` using `rte` clauses)**.**

```
(defun F3 (object)
  (typecase object
    ((rte (:cat symbol (:+ (eql :count) integer)))
     (destructuring-bind (name &key (count 0)) object
        ...))
    ((rte (:cat symbol list (:* string)))
     (destructuring-bind (name data
                          &rest strings) object
        ...))))
```

Notice in the code above that each `rte` clause of the `typecase` includes a call to `destructuring-bind` which is related and hand coded for consistency. The function `F3` is implemented such that the object being destructured is certain to be of the format expected by the corresponding destructuring lambda list.

We provide a macro `destructuring-case` which combines the capability of `destructuring-bind` and `typecase`. Moreover, `destructuring-case` constructs the `rte` type specifiers in an intelligent way, taking into account not only the structure of the destructuring lambda list but also any given type declarations.

Example 4.19 (Using `destructuring-case` including type discrimination)**.**

```
(defun F4 (object)
  (destructuring-case object
    ((name &key count)
     (declare (type symbol name)
              (type integer count))
     ...)
    ((name data &rest strings)
     (declare (type name symbol)
              (type data list)
              (type strings
                    (rte (:* string))))
     ...)))
```

This macro is able to parse any valid destructuring lambda list and convert it to a regular type expression. Supported syntax includes `&whole`, `&optional`, `&key`, `&allow-other-keys`, `&aux`, and recursive lambda lists such as:

Example 4.20 (Lambda list with multiple lambda list keywords)**.**

```
(&whole llist a (b c)
 &key x ((:y (c d)) '(1 2))
 &allow-other-keys)
```

An important feature of `destructuring-case` is that it can construct regular type expressions much more complicated than would be practical by hand. Consider the following example which includes two destructuring lambda lists, whose computed regular type expressions pretty-print to about 20 lines each. An example of the regular type expression matching `Case-1` of Example 4.21 is shown in Example 4.22. The regular type expression for `Case-2` of Example 4.21 is shown in Example 4.23.

Example 4.21 (Using `destructuring-case` with complex lambda lists)**.**

```
(destructuring-case data

  ;; Case-1
  ((&whole llist
     a (b c)
     &rest keys
     &key x y z
     &allow-other-keys)
   (declare (type fixnum a b c)
            (type symbol x)
            (type string y)
            (type list z))
   ...)

  ;; Case-2
  ((a (b c)
   &rest keys
   &key x y z)
   (declare (type fixnum a b c)
            (type symbol x)
            (type string y)
            (type list z))
   ...))
```

Example 4.22 (Regular type expression matching destructuring lambda list Case-1 of Example 4.22)**.**

```
(:cat (:cat fixnum (:and list (rte (:cat fixnum fixnum))))
    (:and
      (:* keyword t)
      (:or
        (:cat (:? (eql :x) symbol (:* (not (member :y :z)) t))
              (:? (eql :y) string (:* (not (eql    :z))    t))
              (:? (eql :z) list   (:* t t)))
        (:cat (:? (eql :y) string (:* (not (member :x :z)) t))
              (:? (eql :x) symbol (:* (not (eql    :z))    t))
              (:? (eql :z) list   (:* t t)))
        (:cat (:? (eql :x) symbol (:* (not (member :y :z)) t))
              (:? (eql :z) list   (:* (not (eql    :y))    t))
              (:? (eql :y) string (:* t t)))
        (:cat (:? (eql :z) list   (:* (not (member :x :y)) t))
              (:? (eql :x) symbol (:* (not (eql    :y))    t))
              (:? (eql :y) string (:* t t)))
        (:cat (:? (eql :y) string (:* (not (member :x :z)) t))
              (:? (eql :z) list   (:* (not (eql    :x))    t))
              (:? (eql :x) symbol (:* t t)))
        (:cat (:? (eql :z) list   (:* (not (member :x :y)) t))
              (:? (eql :y) string (:* (not (eql    :x))    t))
              (:? (eql :x) symbol (:* t t))))))
```

Example 4.23 (Regular type expression matching destructuring lambda list Case-2 of Example 4.22)**.**

```
(:cat (:cat fixnum (:and list (rte (:cat fixnum fixnum))))
      (:and (:* keyword t)
        (:or
         (:cat (:? (eql :x) symbol (:* (eql :x)          t))
               (:? (eql :y) string (:* (member :y :x)    t))
```

```
       (:? (eql :z) list   (:* (member :z :y :x) t)))
(:cat  (:? (eql :y) string (:* (eql    :y)       t))
       (:? (eql :x) symbol (:* (member :x :y)    t))
       (:? (eql :z) list   (:* (member :z :x :y) t)))
(:cat  (:? (eql :x) symbol (:* (eql    :x)       t))
       (:? (eql :z) list   (:* (member :z :x)    t))
       (:? (eql :y) string (:* (member :y :z :x) t)))
(:cat  (:? (eql :z) list   (:* (eql    :z)       t))
       (:? (eql :x) symbol (:* (member :x :z)    t))
       (:? (eql :y) string (:* (member :y :x :z) t)))
(:cat  (:? (eql :y) string (:* (eql    :y)       t))
       (:? (eql :z) list   (:* (member :z :y)    t))
       (:? (eql :x) symbol (:* (member :x :z :y) t)))
(:cat  (:? (eql :z) list   (:* (eql    :z)       t))
       (:? (eql :y) string (:* (member :y :z)    t))
       (:? (eql :x) symbol (:* (member :x :y :z) t))))))
```

Examples 4.22 and 4.23 show two machine generated regular type expressions. The DFAs of those regular type expressions are illustrated in Figures 4.6 and 4.7 respectively. The two DFAs have striking similarities. This is not surprising as the lambda lists from the `destructuring-case` which generates them differ only by the inclusion of `&allow-other-keys` in `Case-1` of Example 4.21. Figure 4.5 itemizes the labels for the transitions in Figures 4.6 and 4.7.

Transition label	**Regular type expression**
T_1	`t`
T_2	`list`
T_3	`fixnum`
T_4	`symbol`
T_5	`keyword`
T_6	`string`
T_7	`(and list (rte (:cat fixnum fixnum)))`
T_8	`(eql :x)`
T_9	`(eql :y)`
T_{10}	`(eql :z)`
T_{11}	`(member :x :y)`
T_{12}	`(member :x :z)`
T_{13}	`(member :y :z)`
T_{14}	`(member :x :y :z)`
T_{15}	`(and keyword (not (eql :x)))`
T_{16}	`(and keyword (not (eql :y)))`
T_{17}	`(and keyword (not (eql :z)))`
T_{18}	`(and keyword (not (member :x :y)))`
T_{19}	`(and keyword (not (member :x :z)))`
T_{20}	`(and keyword (not (member :y :z)))`

Figure 4.5: Transition types for Case-1 Figure 4.6 and Case-2 Figure 4.7

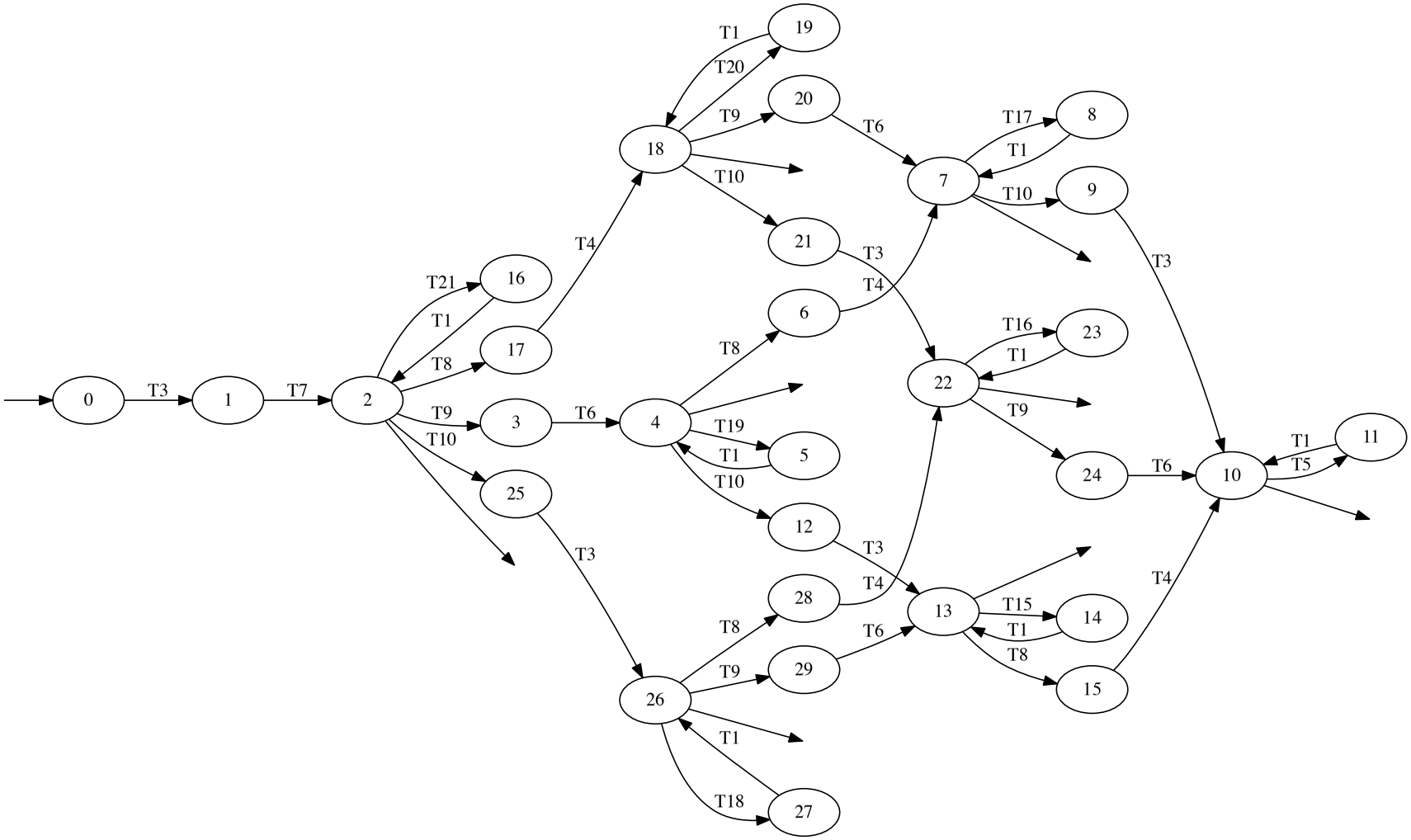


Figure 4.6: DFA recognizing the destructuring lambda list Case-1

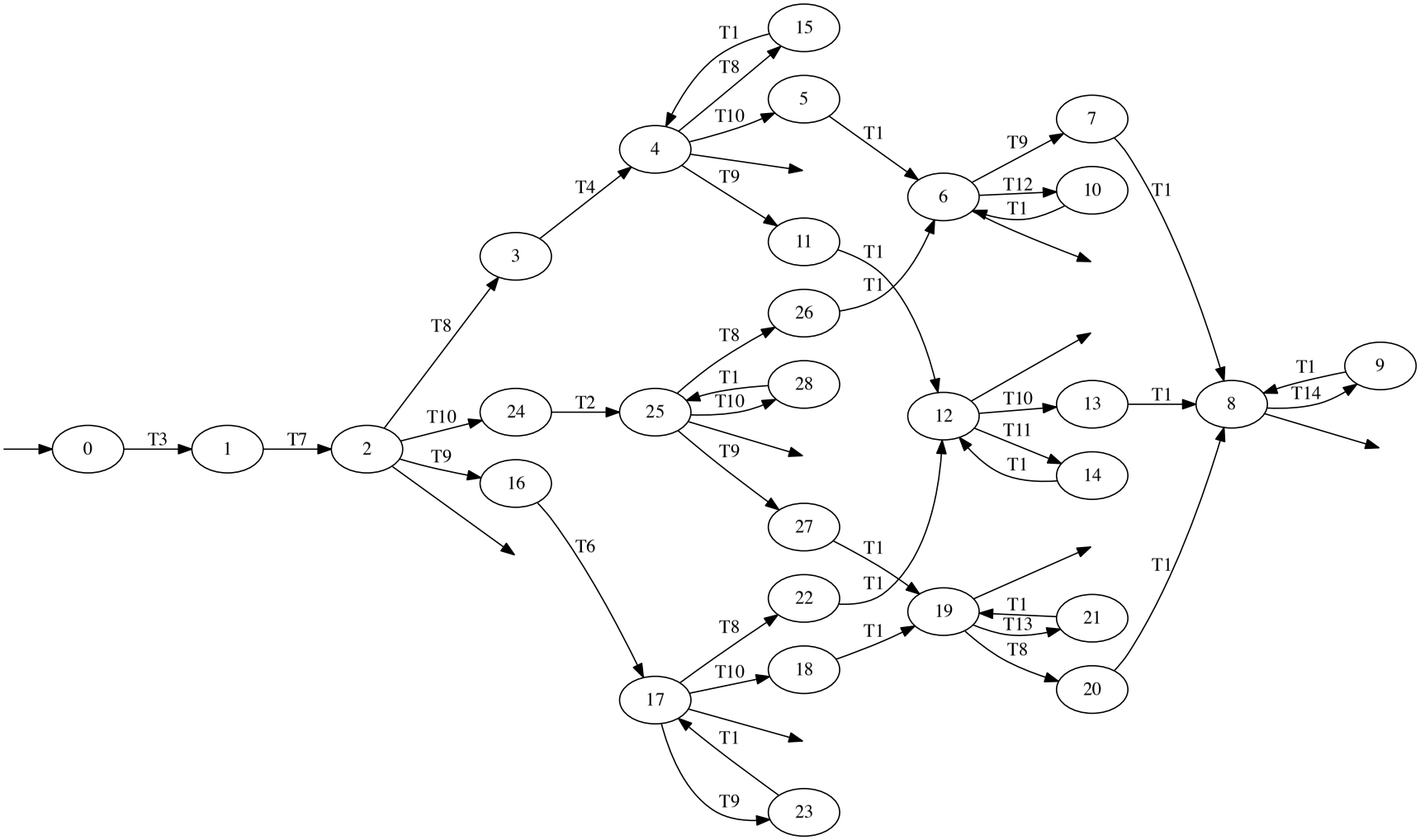


Figure 4.7: DFA recognizing the destructuring lambda list Case-2

There is a caveat to be aware of when using `destructuring-case`. We do not attempt to solve the problem presented if the actual type of the default value of an optional argument does not match the declared type. We believe this problem to be unsolvable in general, because the extreme case is equivalent to the halting problem. However, it could be solved for a wide range of special cases. An attempt at a partial solution might make it more confusing, as the user would be able to easily know if his case is one of those special cases.

Example 4.24 (Example destructuring-case use case)**.**

```
(destructuring-case '(42)
  ((a &key (count (foo))) ; case-1
   (declare (type number a count))
   ...)
  ((a)   ; case-2
   (declare (type number a))
   ...))
```

Example 4.24 shows the general unsolvable case. Here the default value for the `count` key variable is the return value of the function `foo`. The issue is that we cannot know whether `foo` will return a value which is of type `number` as per the declaration. If `foo` returns a `number`, the `list (42)` matches the first destructuring clause, case-1; otherwise `(42)` matches the second destructuring clause, case-2. We cannot know this by examining the given data `(42)`, and we cannot build a state machine (nor any algorithm) which can make this decision without calling the function `foo`, and thus suffering its side-effects, even if it turns out to not match.

We could, however implement some very common special cases, but we are not sure doing so would enhance the general usability.

Take the simplest special case for example, the case where no explicit default is specified for the `&key` variables (similar for `&optional` variables). We know that the default value in this case is specified as `nil`, and we know that `nil` is not of type number. Thus in `(42)` does not match case-1.

Example 4.25, if `DATA='(42)`, then case-2 is satisfied. If `DATA='(42 :count 3)` then case-1 is satisfied.

Example 4.25 (Special case of destructuring-case)**.**

```
(destructuring-case DATA
  ((a &key count) ; case-1
   (declare (type number a count))
   ...)
  ((a)   ; case-2
   (declare (type number a))
   ...))
```

Similarly if there *IS* a default given which is a literal (literal string, quoted symbol, number, *etc.*) we can figure out (at compile time) whether that literal matches the declared value, in order to determine whether `:count` is actually required or optional in the destructured list.

Example 4.26, if `DATA='(42)`, then case-2 is satisfied. If `DATA='(42 :count 3)` then case-1 is satisfied. Case-3 is redundant.

Example 4.26 (Special case of destructuring-case)**.**

```
(destructuring-case '(42)
  ((a &key (count "hello")) ; case-1
   (declare (type number a)
            (type string count))
   ...)
```

```
  ((a &key (count 0)) ; case-2
   (declare (type number a count))
   ...)
  ((a)   ; case-3
   (declare (type number a))
   ...))
```

If the default for the `&key` variable is a symbol which is declared a constant, it reduces to the special case in Example 4.26. However, it is unclear whether it is possible to know at macro expansion time whether a symbol names a constant.

Example 4.27, if `DATA='(42)`, then case-1 is satisfied. Case-2 is redundant.

Example 4.27 (Special case of destructuring-case)**.**

```
(defconstant +ZERO+ 0)

(destructuring-case '(42)
  ((a &key (count +ZERO+)) ; case-1
   (declare (type number a count))
   ...)
  ((a)   ; case-2
   (declare (type number a))
   ...))
```

For these reasons we don't attempt to implement any of these special cases. One additional argument is that SBCL warns against such usage anyway. Example 4.28 shows warnings issued at compile time that the default value, `nil`, does not match the declared type. The same warning appears in the corresponding `destructuring-case` because it expands to `destructuring-bind`.

Example 4.28 (Warnings from dubious destructuring-bind)**.**

```
(destructuring-bind (a &key count ) DATA
  (declare (type number count))
  ...)

; in: DESTRUCTURING-BIND (A &KEY COUNT)
;     (IF (NOT (EQL #:G685 0))
;         (CAR (TRULY-THE CONS #:G685)))
; ==>
;   NIL
;
; caught STYLE-WARNING:
;   The binding of COUNT is not a NUMBER:
;     NIL
;   See also:
;     The SBCL Manual, Node "Handling of Types"
```

4.4 Implementation overview

Using an `rte` involves several steps. The following subsections describe these steps.

1. Convert a parameterized `rte` type into code that will perform run-time type checking.

2. Convert the regular type expression to DFA (deterministic finite automaton, sometimes called a *finite state machine*).

3. Decompose a list of type specifiers into disjoint types.

4. Convert the DFA into code which will perform run-time execution of the DFA.

4.4.1 Pattern matching a sequence

The function `match-sequence`, can be used to determine whether a given sequence matches a given pattern.

Implementation 4.29 (`match-sequence`)**.**

```
(defun match-sequence (input-sequence pattern)
  (declare (type list pattern))
  (when (typep input-sequence 'sequence)
    (let ((sm (or (find-state-machine pattern)
                  (remember-state-machine (make-state-machine pattern)
                                          pattern))))
      (some #'state-final-p
            (perform-transitions sm input-sequence)))))
```

This function takes an input sequence such as a `list` or `vector`, and a regular type expression, and returns `true` or `false` depending on whether the sequence matches the regular type expression. It works as follows:

1. If necessary it builds a finite state machine by calling `make-state-machine`, and caches it to avoid having to rebuild the state machine if the same pattern is used again.

2. Next, it executes the machine according to the input sequence.

3. Finally, it asks whether any of the returned states are final states.

4.4.2 Type definition

The `rte` type is defined by `deftype`.

Implementation 4.30 (`deftype rte`)**.**

```
(deftype rte (pattern)
  `(and sequence
        (satisfies ,(compute-match-function
                     pattern))))
```

As in this definition, when the `satisfies` type specifier is used, it must be given a `symbol` naming a globally callable unary function. In our case `compute-match-function` accepts a regular type expression, such as `(:cat number (:* string))`, and computes a named unary predicate. The predicate can thereafter be called with a sequence and will return `true` or `false` indicating whether the sequence matches the pattern. Notice that the pattern is usually provided at *compile*-time, while the sequence is provided at *run*-time. Furthermore, `compute-match-function` ensures that given two patterns which are `EQUAL`, the same function name will be returned, but will only be created and compiled once.

The definition of the `rte` parameterized type is a bit more complicated that we'd like. We'd actually like to define it as a type using `satisfies` of a function which closes over, or even embeds the given pattern, but neither of these is possible. In fact the argument of `satisfies` must be a symbol naming a global function; a function object is not accepted as argument of `satisfies`. Example 4.31 shows two invalid definitions.

Example 4.31 (Invalid `rte` type definitions)**.**

```
;; first INVALID type definition
(deftype rte (pattern)
  `(and sequence
        (satisfies ,(lambda (input-sequence)
                      (match-sequence input-sequence pattern)))))

;; second INVALID type definition
(deftype rte (pattern)
  `(and sequence
        (satisfies `(lambda (input-sequence)
                      (match-sequence input-sequence ,pattern)))))
```

The `rte` type definition, from Implementation 4.30, has a return value (a type expansion) and several side effects.

1. creates an *intermediate function* which closes over the given pattern
   `(lambda (input-sequence) (match-sequence input-sequence pattern))`
2. creates a function name unique for the given pattern.
3. uses `(setf symbol-function)` to define a function whose function binding is that intermediate function
4. the `deftype` expands to `(and sequence (satisfies` *`that-function-name`*`))`

Example 4.32 (3-D point type definition)**.**

```
(deftype 3-d-point ()
  `(rte (:cat number number number)))
```

Example 4.32 should make it clearer. The type `3-d-point` invokes the `rte` parameterized type definition with argument `(:cat number number number)`. The `deftype` of `rte` assures that a function is defined as in Implementation 4.33.

Implementation 4.33.

```
(defun rte::|(:cat number number number)|
    (input-sequence)
  (match-sequence input-sequence
                  '(:cat number number number)))
```

The function name, `|(:cat number number number)|` even if somewhat unusual, is so chosen to improve the error message and back-trace that occurs in some situations. The `|...|` notation is the Common Lisp reader syntax to denote a symbol containing spaces or other delimiters characters. *E.g.*, `|(a b)|` is a symbol whose print-name is "(a b)". The following back-trace occurs when attempting to evaluate a failing assertion.

Example 4.34 (Error message on failed `rte` type assertion)**.**

```
CL-USER> (the 3-d-point (list 1 2))

The value (1 2)
is not of type
  (OR (AND #1=(SATISFIES |(:cat NUMBER NUMBER NUMBER)|)
           CONS)
      (AND #1# NULL) (AND #1# VECTOR)
      (AND #1# SB-KERNEL:EXTENDED-SEQUENCE)).
   [Condition of type TYPE-ERROR]

Restarts:
 0: [RETRY] Retry SLIME REPL evaluation request.
 1: [*ABORT] Return to SLIME's top level.
 2: [ABORT] abort thread (#<THREAD "repl-thread" RUNNING {1012A80003}>)

Backtrace:
  0: ((LAMBDA ()))
  1: (SB-INT:SIMPLE-EVAL-IN-LEXENV (THE 3-D-POINT (LIST 1 2)) #<NULL-LEXENV>)
  2: (EVAL (THE 3-D-POINT (LIST 1 2)))
 --more--
```

It is also assured that the DFA corresponding to `(:cat number number number)` is built and cached, to avoid unnecessary re-creation at run-time. Finally, the type specifier `(rte (:cat number number number))` expands to the following.

Example 4.35 (Expansion of `rte` type specifier)**.**

```
(and sequence
     (satisfies |(:cat number number number)|))
```

4.4.3 Constructing a DFA representing a regular type expression

In order to determine whether a given sequence matches a particular regular type expression, we conceptually execute a DFA with the sequence as input. Thus we must convert the regular type expression to a DFA. This need only be done once and can often be done at compile time. The flow involves the Brzozowski algorithm, explained in Section 3.6.

The set of sequences of Common Lisp objects is not a rational language, because for one reason, the perspective alphabet (the set of all possible Common Lisp objects) is not a finite set.[1] Even though the set of sequences of objects is infinite, the set of sequences of type specifiers is a rational language, if we only consider as the alphabet, the set of type specifiers explicitly referenced in a regular type expression. With this choice of alphabet, sequences of Common Lisp type specifiers conform to the definition of *words* in a *rational language*.

There is a delicate matter when mapping a sequence of objects to a sequence of type specifiers: the mapping is not unique, and may lead to a non-deterministic finite automata (NDFA). We would like to avoid NDFAs, because the algebra associated with DFAs is simpler, especially when involving intersection and complementation. We ignore this issue for the moment, but return to it Section 4.4.6.

Consider the rational type expression

$$P_0 = (symbol \cdot (number^+ \cup string^+))^+ . \tag{4.9}$$

We wish to construct a DFA which recognizes sequences matching this pattern. Such a DFA is shown in Figure 4.8. As the first step in this construction, we create a state P_0 corresponding to the given rational type expression, (4.9).

Next, we proceed to calculate the derivative with respect to each type specifier mentioned in P_0. Actually, as will be seen, it suffices to differentiate with respect to the type specifiers which are permissible as the first

[1] The computation model of Common Lisp assumes infinite memory. In reality the memory is finite, but as far as theoretical considerations we assume the memory, and thus the set of all potential objects is infinite.

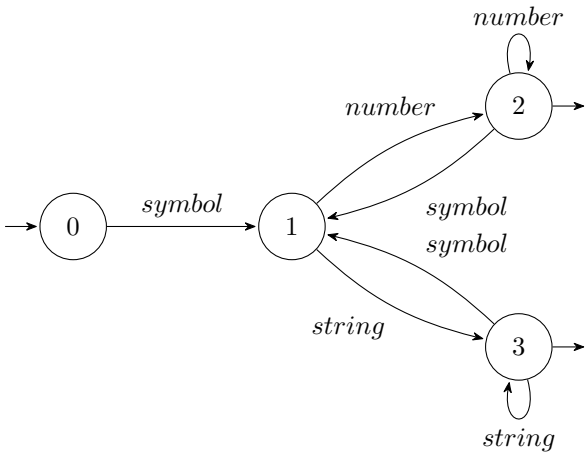


Figure 4.8: Example DFA implementing the rational type expression: $(symbol \cdot (number^+ \cup string^+))^+$

element of the sequence. For example, the first element of the sequence is neither allowed to be a `string` nor a `number`. This is equivalent to saying that the corresponding derivatives are $\varnothing$.

$$\begin{aligned} \partial_{string} P_0 &= \varnothing \\ \partial_{number} P_0 &= \varnothing \end{aligned}$$

Thus, as far as P_0 is concerned, we need only calculate one derivative: $\partial_{symbol} P_0$. To help understand this calculation, we show the steps explicitly.

$$\begin{aligned} \partial_{symbol} P_0 &= \partial_{symbol}\big((symbol \cdot (number^+ + string^+))^+\big) && \text{By (4.9)} \\ &= \partial_{symbol}\big(symbol \cdot (number^+ + string^+)\big) \\ &\quad \cdot \big(symbol \cdot (number^+ + string^+)\big)^* && \text{By Theorem 3.30} \\ &= \big(\partial_{symbol} symbol \cdot (number^+ + string^+) \\ &\quad + \nu(symbol) \cdot \partial_{symbol}(number^+ + string^+)\big) \\ &\quad \cdot \big(symbol \cdot (number^+ + string^+)\big)^* && \text{By (3.14)} \\ &= \big(\varepsilon \cdot (number^+ + string^+) + \varnothing \cdot \partial_{symbol}(number^+ + string^+)\big) \\ &\quad \cdot (symbol \cdot (number^+ + string^+))^* && \text{By (3.10) and (3.3)} \\ P_1 &= (number^+ + string^+) \cdot \big(symbol \cdot (number^+ + string^+)\big)^* && (4.10) \end{aligned}$$

Since there is not yet a state in the automaton labeled with this expression, we create one named P_1. We also create a transition $P_0 \xrightarrow{symbol} P_1$. This transition is labeled with *symbol* because $\partial_{symbol} P_0 = P_1$. The transition corresponds to the arrow on the graph in Figure 4.8 from P_0 to P_1 labeled *symbol*. We now proceed to calculate the derivatives of P_1.

$$\begin{aligned} P_2 &= \partial_{number} P_1 \\ &= \partial_{number}\big((number^+ + string^+) \cdot \big(symbol \cdot (number^+ + string^+)\big)^*\big) && \text{By (4.10)} \\ &= number^* \cdot (symbol \cdot (number^+ + string^+))^* && (4.11) \end{aligned}$$

We add a state P_2 to the automaton with a transition $P_1 \xrightarrow{number} P_2$.

$$\begin{aligned} P_3 &= \partial_{string} P_1 \\ &= string^* \cdot (symbol \cdot (number^+ + string^+))^* && (4.12) \end{aligned}$$

We add a state P_3 to the state machine with a transition $P_1 \xrightarrow{string} P_3$.

If we continue calculating the derivatives, we find that we again obtain expression we have already obtained in earlier steps.

$$\begin{aligned}\partial_{symbol} P_1 &= P_1\\ \partial_{number} P_2 &= P_2\\ \partial_{string} P_3 &= P_3\\ \partial_{symbol} P_2 &= P_1\\ \partial_{symbol} P_3 &= P_1\end{aligned}$$

From these derivatives we create the following transitions thus completing the transitions in the state machine (Figure 4.8): $P_1 \xrightarrow{symbol} P_1$, $P_2 \xrightarrow{number} P_2$, $P_3 \xrightarrow{string} P_3$, $P_2 \xrightarrow{symbol} P_1$, and $P_3 \xrightarrow{symbol} P_1$.

Now, we have created all the states in the automaton, and we have labeled all the transitions between states with the type specifier which was used in the derivative calculation between those states. We have ignored transitions from any state to the $\varnothing$ state.

Finally, we label the *final* states. To determine which states are final states, we must determine which of the rational expressions are nullable. The final states are P_2 and P_3 because only those two states are nullable. We know this intuitively, because

$$\varepsilon \in (number^* \cdot \ (symbol \cdot (number^+ \ \cup \ string^+))^*)$$

can match the empty sequence, and so can

$$\varepsilon \in (string^* \cdot (symbol \cdot (number^+ \ \cup \ string^+))^*)\,.$$

$$\begin{aligned}
\nu(P_0) &= \nu((symbol \cdot (number^+ + string^+))^+) && \text{By (4.9)}\\
&= \nu(symbol \cdot (number^+ + string^+)) && \text{By Theorem 3.32}\\
&= \varnothing \cap \nu(number^+ + string^+) && \text{By (3.5)}\\
&= \varnothing && \text{So } P_0 \text{ is not nullable.}\\
\nu(P_1) &= \nu((number^+ + string^+) \cdot (symbol \cdot (number^+ + string^+))^*) && \text{By (4.10)}\\
&= \nu(number^+ + string^+) \cap \nu((symbol \cdot (number^+ + string^+))^*) && \text{By (3.5)}\\
&= \nu(number^+ + string^+) \cap \varepsilon && \text{By (3.7)}\\
&= \nu(number^+ + string^+) && \text{By Lemma 3.31}\\
&= \nu(number^+) + \nu(string^+) && \text{By (3.4)}\\
&= \nu(number) + \nu(string) && \text{By Theorem (3.32)}\\
&= \varnothing + \varnothing && \text{By (3.3)}\\
&= \varnothing && \text{So } P_0 \text{ is not nullable.}
\end{aligned}$$

In similar manner we find that no other of the expressions is nullable.

$$\begin{aligned}
\nu(P_2) &= \nu\big(number^* \cdot (symbol \cdot (number^+ + string^+))^*\big) && \text{By (4.11)}\\
&= \nu(number^*) \cap \nu\big((symbol \cdot (number^+ + string^+))^*\big) && \text{By (3.5)}\\
&= \varepsilon \cap \varepsilon && \text{By (3.7)}\\
&= \varepsilon && \text{So } P_2 \text{ is nullable.}\\
\nu(P_3) &= \nu\big(string^* \cdot (symbol \cdot (number^+ + string^+))^*\big) && \text{By (4.12)}\\
&= \nu(string^*) \cap \nu\big((symbol \cdot (number^+ + string^+))^*\big) && \text{By (3.5)}\\
&= \varepsilon \cap \varepsilon && \text{By (3.7)}\\
&= \varepsilon && \text{So } P_3 \text{ is nullable.}
\end{aligned}$$

4.4.4 Optimized code generation

In section 4.4.1 we saw a general purpose implementation of an NDFA (non-deterministic finite state machine). There are several techniques which can be used to improve the run-time performance of this algorithm. First we discuss some of the optimizations we have made, and in section 4.4.9 there is a discussion of the performance results.

One thing to notice is that although the implementation described above is general enough to support non-deterministic state machines, the development made in sections 4.4.6 and Chapters 8 and 9 obviate the need for this flexibility. In fact although each state in a state machine recognizing a rational type expression has multiple transitions to next states, we have assured that maximally one such is ever valid as each transition is labeled with a type disjoint from the other transitions from the same state. The result is that to make a state transition in the DFA case, type membership tests must be made only until one is found which matches, whereas in the NDFA case all type membership tests must be made from each state, and a list of matching next states must be maintained.

Another thing to notice is that rather than traversing the state machine to match an input sequence, we may rather traverse the state machine to produce code which can later match an input sequence. The generated code will be special purpose and will only be able to match a sequence matching the particular regular type expression. There are three obvious advantages of the code generation approach. 1) There will be much less code per regular type expression to execute at run time, that code being specifically generated for the specific pattern we are attempting to match. 2) We can avoid several function calls in the code by making use of `tagbody` and `go`. 3) The Lisp compiler can be given a chance to optimize the more specific (less generic) code.

The result of these three considerations is that the code no longer makes use of the potentially costly call to `match-sequence`. In its place, code is inserted specifically checking the regular type expression in question. Implementation 4.36 shows a sample body of such a function which recognizes the regular type expression `((:+ (:cat symbol (:or (:+ number) (:+ symbol)))))`. The corresponding rational type expression is $(symbol \cdot (number^+ + string^+))^+$. The DFA can be seen in Figure 4.8. The code contains two sections, one for the case that the given sequence, `sequence` is a `list` and another if the sequence is a `vector`. The purpose of the two sections is so that the generated code may use more specific accessors to iterate through the sequence, and also so the compiler can have more information about the types of structure being accessed.

Each section differs in how it iterates through the sequence and how it tests for end of sequence, but the format of the two sections is otherwise the same. Each section contains one label for each state in the state machine. Each transition in the DFA is represented as a branch of a `typecase` and `(go ...)` (the Common Lisp `GOTO`).

Implementation 4.36 (Machine generated pattern matcher for recognizing an RTE)**.**

```
(lambda (sequence)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (block check
    (typecase sequence  ; OUTER-TYPECASE
      (list
       (tagbody
        0
          (when (null sequence)
            (return-from check nil)) ; rejecting
          (typecase (pop sequence) ; INNER-TYPECASE
            (symbol (go 1))
            (t (return-from check nil)))
        1
          (when (null sequence)
            (return-from check nil)) ; rejecting
          (typecase (pop sequence) ; INNER-TYPECASE
            (number (go 2))
            (string (go 3))
            (t (return-from check nil)))
        2
          (when (null sequence)
            (return-from check t)) ; accepting
          (typecase (pop sequence) ; INNER-TYPECASE
            (number (go 2))
            (symbol (go 1))
            (t (return-from check nil)))
        3
          (when (null sequence)
            (return-from check t)) ; accepting
          (typecase (pop sequence) ; INNER-TYPECASE
            (string (go 3))
```

```
        (symbol (go 1))
        (t (return-from check nil)))))
  (simple-vector ...)
  (vector        ...)
  (sequence      ...)
  (t nil))))
```

The mechanism we chose for implementing the execution (as opposed to the generation) of the DFA was to generate specialized code based on `typecase`, `tagbody`, and `go`. As an example, consider the DFA shown in Figure 4.8. The code in Implementation 4.36 was generated given this DFA as input.

The code is organized according to a regular pattern. The `typecase`, commented as `OUTER-TYPECASE` switches on the type of the sequence itself. Whether the sequence, `sequence`, matches one of the carefully ordered types `list`, `simple-vector`, `vector`, or `sequence`, determines which functions are used to access the successive elements of the sequence: `svref`, `incf`, `pop`, *etc.*

The final case, `sequence`, is especially useful for applications which wish to exploit the SBCL feature of *Extensible sequences* [New15, Section 7.6] [Rho09]. One of our `rte` based applications uses extensible sequences to view vertical and horizontal *slices* of 2D arrays as sequences in order to match certain patterns within row vectors and column vectors.

While the code is iterating through the sequence, if it encounters an unexpected end of sequence, or an unexpected type, the function returns `nil`. These cases are commented as `rejecting`. Otherwise, the function will eventually encounter the end of the sequence and return `t`. These cases are commented `accepting` in the figure.

Within the inner section of the code, there is one label per state in the state machine. In the example, the labels P_0, P_1, P_2, and P_3 are used, corresponding to the states in the DFA in Figure 4.8. At each step of the iteration, a check is made for end-of-sequence. Depending on the state either `t` or `nil` is returned depending on whether that state is a final state of the DFA or not.

The next element of the sequence is examined by the `INNER-TYPECASE`, and depending on the type of the object encountered, control is transferred (via `go`) to a label corresponding to the next state.

There is a potential efficiency problem in the order of the clauses of this `typecase`. We return to this issue in Chapter 11.

One thing to note about the complexity of this function is that the number of states encountered when the function is applied to a sequence is equal or less than the number of elements in the sequence. Thus the time complexity is linear in the number of elements of the sequence and is independent of the number of states in the DFA.

In some cases the same type may be specified with either the `rte` syntax or with the Common Lisp native `cons` type specifier. For example, a list of three numbers can be expressed either as `(cons number (cons number (cons number null)))` or as `(rte (:cat number number number))`.

Should the `rte` system internally exploit the `cons` specifier when possible, thus avoiding the creation of finite state machines? We began investigating this possibility, but abandoned the investigation on discovering that it lead to significant performance degradation for long lists. We measured roughly a 5% penalty for lists of length 5. The penalty grew for longer lists: 25% with a list length of 10, 40% with a list length of 20.

4.4.5 Sticky states

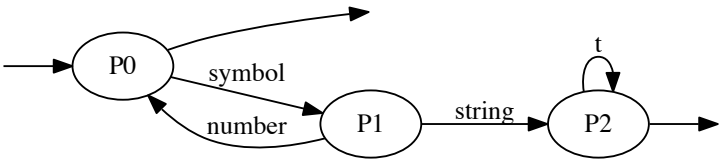


Figure 4.9: DFA with sticky state

Consider the DFA shown in Figure 4.9. If the state machine ever reaches state P_2 it will remain their until the input sequence is exhausted, because the only transition is for the type t, and all objects of this type. This state is called a *sticky state*. If the state machine ever reaches a sticky state which is also a final state, it is no longer necessary to continue examining the input string. The matching function can simply return `true`.

This type of pattern is fairly common such as `(:cat (:* symbol number) (:* t))`.

We have incorporated this optimization into both the generic DFA version (based on `matchsequence`) and also the auto-generated code version. To understand the consequence of this optimization consider a list of length 1000 which begins with a `symbol` followed by a `number`. With the sticky state optimization, checking the pattern against the sequence would involve:

- one check of `(typep object symbol)`,
- one check of `(typep object number)`, and
- 998 checks of `(typep object t)`, all of which are sure to return `true`.

When this optimization is in effect, 1000 type checks are reduced to 2 type checks.

4.4.6 The overlapping types problem

In the example in Section 4.4.3, all the types considered (`symbol`, `string`, and `number`) were disjoint. If the same method is naïvely used with types which are intersecting, the resulting DFA will not be a valid representation of the rational expression. Consider the rational expression involving the intersecting types *integer* and *number*:

$$P_0 = \big((number \cdot integer) \ \cup \ (integer \cdot number)\big) .$$

The sequences which match this expression are sequences of two numbers, at least one of which is an integer. Unfortunately, when we calculate $\partial_{number} P_0$ and $\partial_{integer} P_0$ we arrive at a different result.

$$\begin{aligned} rcl\partial_{number} P_0 &= \partial_{number}\big((number \cdot integer) + (integer \cdot number)\big) \\ &= \partial_{number}(number \cdot integer) \cup \partial_{number}(integer \cdot number) \\ &= (\partial_{number} number) \cdot integer \cup (\partial_{number} integer) \cdot number \\ &= \varepsilon \cdot integer \cup \varnothing \cdot number \\ &= integer \cup \varnothing \\ &= integer \end{aligned}$$

Without continuing to calculate all the derivatives, it is already clear that this result is wrong. If we start with the set of sequences of two numbers one of which is an integer, and out of that find the subset of sequences starting with a number, we get back the entire set. The set of suffixes of this set is not the set of singleton sequences of integer as this naïve calculation of $\partial_{number} P_0$ predicts.

The problem is that if a rational type expression is treated blindly as an ordinary rational expression, then $number \neq integer$ end of story. But if we wish to create a DFA which will allow validation of Common Lisp sequences of objects, we must extend the theory slightly to accommodate intersecting types.

To address this problem, we augment the algorithm of Brzozowski with an additional step. Rather than calculating the derivative at each state with respect to each type mentioned in the regular type expression, some of which might be overlapping, instead we calculate a disjoint set of types. More specifically, given a set $\mathcal{A}$ of potentially overlapping types, we calculate a set $\mathcal{B}$ which has the properties: Each element of $\mathcal{B}$ is a subtype of some element of $\mathcal{A}$, any two elements $\mathcal{B}$ are disjoint from each other, and $\cup\mathcal{A} = \cup\mathcal{B}$. We refer to this deconstruction as the Maximal Disjoint Type Decomposition (MDTD) problem, and we discuss it in depth in Chapters 8 and 9.

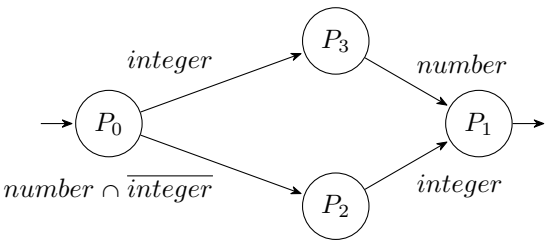


Figure 4.10: Example DFA with disjoint types

Figure 4.10 illustrates a deterministic finite automata (DFA) whose transitions are based on such a disjoint union. The set of overlapping types $\mathcal{A} = \{number, integer\}$ has been replaced with the set of disjoint types $\mathcal{B} = \{number \cap \overline{integer}, integer\}$.

This extra step has two positive effects on the algorithm. 1) it ensures that the constructed automaton is deterministic, *i.e.*, we ensure that all the transitions *leaving* a state specify disjoint types, and 2) it forces our treatment of the problem to comply with the assumptions required by the Brzozowski/Owens algorithm.

The troublesome rule, introduced in Figure 3.3, is equation (3.11). It indicates that $\partial_a b = \varnothing$ for $b \neq a$. The rules in Figure 4.11 show derivatives of type expressions with respect to particular types. Most notably, Figure 4.11 augments Figure 3.3 in the case disjoint types.

$$\partial_A B = \varepsilon \qquad \text{if } A = B \tag{4.13}$$

$$\partial_A B = \varnothing \qquad \text{if } A \cap B = \varnothing \tag{4.14}$$

$$\partial_A B \qquad \text{is undefined otherwise.}$$

Figure 4.11: Rules for derivative of regular type expressions

Proof. Arguments justifying (4.13) and (4.14).

Let B_{seq} be a non empty set of sequences of length one, each of whose first elements is an object of type B_{type}. $\partial_A B$ by definition is a particular possibly empty subset of the set of suffixes of B_{seq}. Call that subset S. Now, $\partial_A B = Suff\{S\}$. Since every element of B_{seq} has length one, every suffix and consequently every element of S has length zero. The unique zero length sequence is denoted ε. Thus $\partial_A B$ is either ε[2] or $\varnothing$. In particular if $S = \varnothing$ then $\partial_A B = \varnothing$; if $S \neq \varnothing$ then $\partial_A B = \varepsilon$. What remains is to determine for which cases (4.13) and (4.14) is S empty.

(4.13) Since $A = B_{type}$, $S = B_{seq}$. Since S is not empty, $\partial_A B = \varepsilon$.

(4.14) $S \subset B_{seq}$ is a set of singleton sequences each of whose element is of type A. B_{seq} is a set of sequences whose first element is of type B_{type}. Since no element of type A is an element of B_{type}, S must be empty. Thus $\partial_A B = \varnothing$.

□

To use these differentiation rules, we note that $\partial_A B$ is undefined when A and B are *partially overlapping*. Practically this means we must only differentiate a given rational expression with respect to disjoint types. Figure 4.10 shows an automaton expressing the rational expression $P_0 = ((number \cdot integer) \cap (integer \cdot number))$ but only using types for which the derivative is defined. $P_3 = \partial_{integer} P_0$ and $P_1 = \partial_{(and\ number\ (not\ integer))} P_0$. Figure 4.10 does show transitions from $P_3 \xrightarrow{number} P_2$ and $P_1 \xrightarrow{integer} P_2$ using intersecting types. This is not, however, a violation of the rules in Figure 4.11 because P_3 and P_1 are different states.

We need an algorithm (in this case implemented in Common Lisp) which takes a list of type specifiers, and computes a list of disjoint sub types, such that union of the two sets of types is the same. *E.g.*, given the list `(integer number)` returns the list `(integer (and number (not integer)))`. Chapter 9 explains how this can be done.

Algorithms for decomposing a set of types into a set of disjoint types are discussed in Chapters 8 and 9. At the inescapable core of each algorithm is the Common Lisp function `subtypep` [Bak92]. This function is crucial not only in type specifier simplification, needed to test equivalence of symbolically calculated Brzozowski derivatives, but also in deciding whether two given types are disjoint. For example, we know that `string` and `number` are disjoint because `(and string number)` is a subtype of `nil`. See Sections 2.3 and 2.4 for computation details of type intersection, type disjointness, and type equivalence and concerns about whether it is at all possible to determine the subtype relation programmatically.

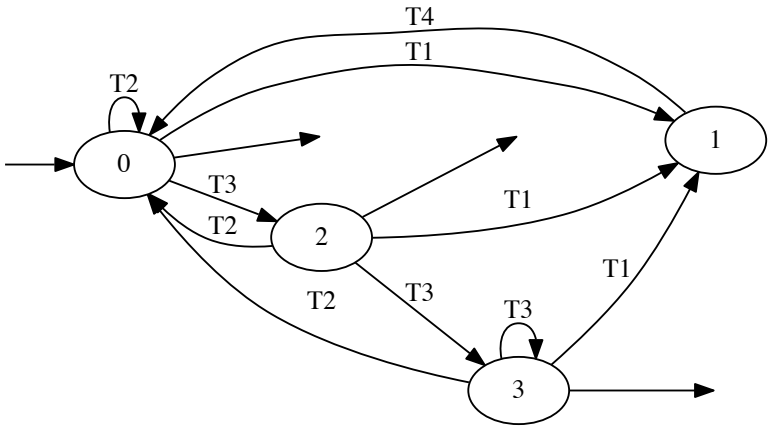
4.4.7 Rational type expressions involving `satisfies`

This section discusses some challenges introduced by the `satisfies` type. Despite the system's inability to peer into the types specified by `satisfies`, we may nevertheless use such types in rational type expressions. Doing so we, get correct but sub-optimal results.

Consider the rational type expression:[3] $((string + odd)^? \cdot even)^*$ which corresponds to the regular type expression:

[2] Recall the abuse of notation that ε denotes both the empty word and the set containing the empty word.

[3] In this case we use the notation of a super-scripted $^?$ to indicate an *optional* expression. Such notation is common in literature relating to regular language theory.



Transition label	Regular type expression
T_1	`(or string (and (satisfies oddp) (not (satisfies evenp))))`
T_2	`(and (not (satisfies oddp)) (satisfies evenp))`
T_3	`(and (satisfies oddp) (satisfies evenp))`
T_4	`(satisfies evenp)`

Figure 4.12: DFA in the case of `satisfies` type

```
(:* (:? (:or string
              (satisfies oddp)))
      (satisfies evenp))
```

The corresponding DFA is shown in Figure 4.12. Although the results are technically correct, they are more complicated than necessary. In particular, transition label T_1, `(or string (and (satisfies oddp) (not (satisfies evenp))))` is equivalent to `(or string (satisfies oddp))`. In addition, consider the transition labels T_2 and T_4, `(and (not (satisfies oddp)) (satisfies evenp))` and `even` respectively. These correspond to the same type.

Furthermore, consider state 3. This state is only reachable via transitions $2 \xrightarrow{T_3} 3$ and $3 \xrightarrow{T_3} 3$. The transition label, T_3 corresponds to type `(and (satisfies oddp) (satisfies evenp))`, which we know is an empty type; no value is both even and odd. Since transition $2 \xrightarrow{T_3} 3$ will never be taken at run-time, state 3 is not accessible, and can thus be eliminated.

We can improve the result. Recall the human knows that `(satisfies oddp)` is not a subtype of `string`, but that the Lisp system does not. The difficulty is that Lisp does not know that `(satisfies oddp)` and `(satisfies evenp)` are non-empty and disjoint. We can improve the situation by defining types, `odd` and `even`, as in Implementation 4.37.

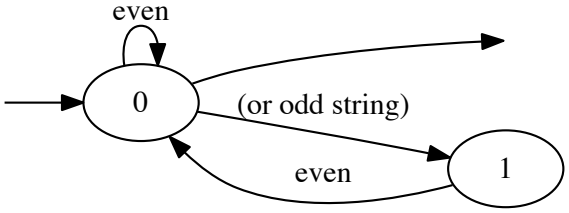


Figure 4.13: DFA with good deftype

Implementation 4.37 (Final version of type definitions of odd and even)**.**

```
(deftype odd ()
  `(and integer
        (not (satisfies evenp))
        (satisfies oddp)))
==> ODD

(deftype even ()
  `(and integer
        (not (satisfies oddp))
        (satisfies evenp)))
==> EVEN

(subtypep 'odd 'even)
==> NIL, T

(subtypep 'odd 'string)
==> NIL, T
```

Given the definitions of the types `even` and `odd` in Implementation 4.37, the `disjoint-types-p` function is able to figure out that types such as `string` and `odd` are disjoint.

With these final type definitions, the state machine representing the expression `(:* (:? (:or string odd)) even)` is shown in Figure 4.13.

It is perhaps worth repeating that the state machines in Figures 4.12 and 4.13 recognize the same sequences. The type specifiers marking the transitions of the former are correct, but less efficient than those in the latter. Additionally, the number of states has been reduced in the latter to two states. However, to achieve this minimal state machine, we supplied redundant information in the type definitions.

4.4.8 Known open issue

With the current state of implementation of `rte` there is a known serious limitation with respect to the compilation semantics. During each expansion of the `rte deftype`, the implementation notices whether this is the first time the given regular type expression has been encountered, and, if so, it creates a named function to check a sequence against the pattern. This flow is explained earlier in Section 4.4. After the named function has been created, the `deftype` expands to something like `(and sequence (satisfies rte::|(:* number)|))`, which is what is written to the fasl file being compiled. So the compiler replaces expression such as `(typep object '(rte (:* number)))` with something like `(typep object '(and sequence (satisfies rte::|(:* number)|)))`. And that's what goes into the fasl file.

The problem occurs the next time Lisp restarts, and the fasl file is loaded. The loader encounters this expression `(typep object '(and sequence (satisfies rte::|(:* number)|)))` and happily reads it. But when the call to `typep` is encountered at run-time, the function `rte::|(:* number)|` is undefined. It is undefined because the closure which was `setf`'ed to the symbol-function existed in the other Lisp image, but not in this one.

Apparently, according to [Pit03], it is a known limitation in the Common Lisp specification. Certain file,s which use `rte` based types, can be compiled but cannot be re-loaded from the compiled file.

If only there were a way to indicate which files should be loaded from source, allowing others to be compiled and loaded from the compiled file. One might think ASDF [Bar15] could be used for this purpose. Unfortunately it probably cannot be. There is no facility in ASDF to mark some files as *load-from-source* and others as *load-from-compiled* [Bar15, Section 16.6.6].

Declaration based solution

In order to force the definition of the *missing* function to be compiled into the fasl file, the declaration macro `defrte` may be used. To use this approach, declare any regular type expression with `defrte` before it appears within a function definition. Moreover, the text of the regular type expression must be EQUAL to the text declared by `defrte`.

Implementation 4.38 (defrte).

```
(defrte (:* number number))

(defun F (a b)
  (declare (type (rte (:* number number)) a b))
  ...)
```

This usage does indeed seem redundant, but is a pretty easy work-around for this insidious problem.

ASDF based solution

This solution allows the ASDF [Bar15] COMPILE-OP operation to create an auxiliary file parallel to the fasl file in the compile directory. The file will have a .rte extension but the same base file name. Later when the fasl file is loaded via an ASDF LOAD-OP operation, the .rte file will be loaded before the fasl file.

There are several steps to follow to effectuate this workaround.

1. Include the :defsystem-depends-on keyword in the asdf:defsystem, to register a dependency on :rte. Use :defsystem-depends-on rather than simply depends-on, otherwise ASDF won't be able to understand the use of :rte-cl-source-file which follows.
2. In the components section, use :file to declare any file which should be compiled and loaded normally, but use :rte-cl-source-file to register a file which contains a problematic regular type expression.

Example 4.39 shows such a defsystem which uses :rte-cl-source-file.

Implementation 4.39 (ASDF based solution using :rte-cl-source-file).

```
(asdf:defsystem :rte-test
  :defsystem-depends-on (:rte)
  :depends-on (:rte-regexp-test
               :2d-array
               (:version :lisp-unit "0.9.0")
               :2d-array-test
               :ndfa-test
               :lisp-types-test)
  :components
  ((:module "rte"
    :components
    ((:file "test-rte")
     (:file "test-list-of")

     ;; CREATE and LOAD a .rte file
     (:rte-cl-source-file "test-re-pattern")

     (:file "test-destructuring-case-1")
     (:file "test-destructuring-case-2")
     (:file "test-destructuring-case")
     (:file "test-ordinary-lambda-list")))))
```

There are a few subtle points with this implementation. The keyword :rte-cl-source-file within the :components section of the ASDF system definition triggers a custom compilation and loading procedure, governed by the CLOS class asdf-user:rte-cl-source-file which inherits directly from asdf:cl-source-file. This class asdf-user:rte-cl-source-file is defined in the :rte package whose loading is triggered by the :defsystem-depends-on (:rte) option in the system definition.

There are two methods specializing on the class asdf-user:rte-cl-source-file shown in Implementation 4.40.

Implementation 4.40 (Methods required by ASDF)**.**

```
(defmethod asdf:perform :around ((operation asdf:compile-op)
                                 (file asdf-user::rte-cl-source-file))
  ...)

(defmethod asdf:perform :before ((operation asdf:load-op)
                                 (file asdf-user::rte-cl-source-file))
   ...)
```

The `asdf:perform :around` method intercepts the `asdf:compile-op` operation to determine which `rte` types and which `rte` patterns get defined by compiling the source file via `(call-next-method)`. Once this list is calculated, the `:around` method writes a `.rte` file along side the fasl file whose text defines pattern definition functions. The `:before` method simply loads this `.rte` from source; *i.e.* the `.rte` file is loaded from source before the fasl is loaded. This procedure guarantees that the functions created as a side effect of compilation are also loaded when the fasl is loaded even if the fasl has already been compiled in another Lisp image.

4.4.9 RTE performance vs hand-written code

A natural question to ask is how the state-machine approach to pattern matching compares to hand written code. That is to say: what is the cost of the declarative approach?

To help answer this question, consider the function `check-hand-written` in Implementation 4.41. It is a straightforward handwritten function to check for a `list` matching the regular type expression `(:* symbol number)`.

Implementation 4.41 (Hand-written pattern checking)**.**

```
(defun check-hand-written (object)
  (or (null object)
      (and (cdr object)
           (symbolp (car object))
           (numberp (cadr object))
           (check-hand-written (cddr object)))))
```

The test we constructed was to attempt to match 200 samples of lists of length 8000. The handwritten code was able to do this in roughly 11ms of CPU time. The generic state machine code to do this took about 879ms, ignoring the initial cost of building the state machine. Using the optimization described in Section 4.4.4, this time dropped to about 22ms.

Version	CPU time (sec)	Penalty
Hand written	0.011	1x
Generic DFA	0.879	77.5x
Generated Code	0.022	2x

4.4.10 RTE performance vs CL-PPCRE

The `rte` type can be used to perform simple string regular expression checking. A generally accepted Common Lisp implementation of regular expressions for strings is `CL-PPCRE` [Wei15].

The following example is similar to the one shown in section 4.3.1.

We would like to count the number of strings in a given list which match a particular regular expression. To analyze the performance we used two approaches: using `CL-PPCRE` and `rte`. In particular, we implemented the following two function `count-matches-ppcre` and `count-matches-rte` respectively shown in Implementation 4.42.

Implementation 4.42 (String regular expression performance testing).

```
(defvar *data* '("abababababzabab"
                 "abababababzababababab"
                 "abababababzababababab"
                 "abababababzzzzababababab"
                 "abababababababzzzzzzzababababab"
                 "abababababababababababzzzzzzzababababab"
                 "abababababababababababababababzzzzzzzababababab"
                 "abababababzzzzzzababababababababzzzzzzzababababab"
                 ))

(defvar *test-scanner*  (cl-ppcre:create-scanner "^(ab)*z*(ab)*$"))

(defun count-matches-ppcre ()
  (count-if (lambda (str)
              (cl-ppcre:scan *test-scanner* str))
            *data*))

(defun count-matches-rte ()
  (count-if (lambda (str)
              (typep str `(rte ,(regexp-to-rte "(ab)*z*(ab)*"))))
            *data*))
```

The performance difference is significant. A loop executing each function one million times, shows that the `rte` approach runs about 35 % faster, as shown in the output in Example 4.43

Example 4.43 (Timing of `rte` vs CL-PPCRE).

```
RTE> (time (dotimes (n 1000000) (rte::count-matches-rte)))
Evaluation took:
  6.185 seconds of real time
  6.149091 seconds of total run time (6.089164 user, 0.059927 system)
  99.42% CPU
  14,808,087,438 processor cycles
  32,944 bytes consed

NIL
RTE> (time (dotimes (n 1000000) (rte::count-matches-ppcre)))
Evaluation took:
  8.425 seconds of real time
  8.334411 seconds of total run time (7.750325 user, 0.584086 system)
  98.92% CPU
  20,172,005,693 processor cycles
  768,016,656 bytes consed

NIL
```

4.4.11 Exceptional situations

In Common Lisp, `defclass` creates a class and a type of the same name. Every valid class name is simultaneously a type specifier. The type is the set of instances of that named class, which includes instances of all sub-classes of the class. Classes can be redefined, especially while an application is being developed and debugged. The implementation described in this report memoizes certain calculations for reuse later. For example, given the rational expression; *i.e.*, the argument list of `rte`, a finite automaton is generated, and cached with the rational expression. The generation of this automaton makes some assumptions about subtype relationships. If classes are redefined later, these relationships may no longer hold; consequently, the memoized automata may no longer be correct.

The Common Lisp Metaobject Protocol [Pae93, KdRB91] (MOP) provides a mechanism, called dependent maintenance protocol, designed to handle this kind of situation. The protocol allows applications to attach *observers* called "dependents" to classes. Thereafter, whenever one of these classes changes, an application specific method is called.

We exploit the dependent maintenance protocol to purge the cache of state machines that have potentially become invalid as a consequence of a CLOS class redefinition.

4.5 Alternatives: Use of cons construct to specify homogeneous lists

One simple and straightforward way to define types representing fixed types of homogeneous lists is illustrated in Example 4.44.

Example 4.44 (Simple attempt to define types)**.**

```
(defun list-of-fixnum (data)
  (every #'(lambda (n) (typep n 'fixnum))
         data))

(deftype list-of-fixnum ()
  `(and list (satisfies list-of-fixnum)))
```

Using this approach the developer could define several types for the various types of lists used the program under development.

The `cons` type construct can be used to declare the types of the `car` and `cdr` of a cons cell, *e.g.*, `(cons number (cons integer (cons string null)))`. The `cons` construct may be used any finite number of times explicitly, *e.g.* `(cons number (cons number (cons number null)))` declares a list of exactly three numbers.

The syntax of the `cons` construct can be tedious, especially for lists of length more than two or three. For example, to specify a list of four numbers, we would use `(cons number (cons number (cons number (cons number null))))`. It is easy to define an intermediate type to simplify the syntax, as show in Implementation 4.45.

Implementation 4.45 (deftype cons*)**.**

```
(deftype cons* (&rest types)
  (cond
    ((cddr types)
     `(cons ,(car types)
            (cons* ,@(cdr types))))
    (t
     `(cons ,@types))))
```

Using the newly defined `cons*` type we can specify a list of four numbers as `(cons* number number number number null)`.

One might ask whether the `rte` implementation might benefit by recognizing lists of fixed length and simply expanding to a Common Lisp type specifier using `cons`. We did indeed consider this question during the development, but found it caused a performance penalty. Admittedly, we only investigated this potential optimization with SBCL, but experimentation showed a roughly 5% penalty for lists of length 5. Moreover, the penalty seems to grow for longer lists: 25% with a list length of 10, 40% with a list length of 20.

Another disadvantage of the approach of using the `cons` specifier is that it is not possible to combine the two approaches above to generalize homogeneous list types of arbitrary length. One might attempt in vain to define a type for homogeneous lists recursively as shown in Implementation 4.46, in order to specify a type such as `(list-of number)`.

Implementation 4.46 (Vain attempt of `deftype list-of`).

```
(deftype list-of (type)
  `(or null (cons ,type (list-of ,type))))
```

But this self-referential type definition is not valid [Mar15], because of the Common Lisp specification of `deftype` which states: *Recursive expansion of the type specifier returned as the expansion must terminate, including the expansion of type specifiers which are nested within the expansion. [Ans94, Section DEFTYPE]*

An attempt to use such an invalid type definition will result in something like the output in Example 4.47.

Example 4.47 (Attempt to use recursive type specifier from Implementation 4.46).

```
CL-USER> (typep (list 1 2 3) '(list-of fixnum))
INFO: Control stack guard page unprotected
Control stack guard page temporarily disabled: proceed with caution

debugger invoked on a SB-KERNEL::CONTROL-STACK-EXHAUSTED in thread
#<THREAD "main thread" RUNNING {1002AEC673}>:
  Control stack exhausted (no more space for function call frames).
This is probably due to heavily nested or infinitely recursive function
calls, or a tail call that SBCL cannot or has not optimized away.

PROCEED WITH CAUTION.

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [ABORT] Exit debugger, returning to top level.

(SB-KERNEL::CONTROL-STACK-EXHAUSTED-ERROR)
0]
```

A caveat of using `rte` is that the usage must obey the restriction explained above, posed by the Common Lisp specification [Ans94, Section DEFTYPE].

As an example of this limitation, in Example 4.48, is a failed attempt to implement a type which matches a unary tree, *i.e.* a type whose elements are `1`, `(1)`, `((1))`, `(((1)))`, *etc.*

Example 4.48 (Attempt to use recursive type definition involving an `rte` type).

```
CL-USER> (deftype unary-tree ()
           `(or (eql 1)
                (rte unary-tree)))
UNARY-TREE
RTE> (typep '(1) 'unary-tree)
Control stack exhausted (no more space for function call
frames).  This is probably due to heavily nested or
infinitely recursive function calls, or a tail call that
SBCL cannot or has not optimized away.

PROCEED WITH CAUTION.
 [Condition of type SB-KERNEL::CONTROL-STACK-EXHAUSTED]
```

4.6 Related work

Attempts to implement `destructuring-case` are numerous. We mention three here. R7RS Scheme provides `case-lambda` [SCG13, Section 4.2.9] which appears to be syntactically similar construct, allowing argument lists of various fixed lengths. However, according to the specification, nothing similar to Common Lisp style destructuring is allowed.

The implementation of `destructuring-case` provided in [Dom] does not have the feature of selecting the clause to be executed according to the format of the list being destructured. Rather it uses the first element of the given list as a case-like key. This key determines which pattern to use to destructure the remainder of the list.

The implementation provided in [Fun13], named `destructure-case`, provides similar behavior to that which we have developed. It destructures the given list according to which of the given patterns matches the list. However, it does not handle destructuring within the optional and keyword arguments as in Example 4.49.

Example 4.49 (`destructuring-case` with keyword arguments)**.**

```
(destructuring-case '(3 :x (4 5))
  ((a &key ((:x (b c))))
   (list 0 a b c)) ;; this clause should be taken
  ((a &key x)
   (list 2 a x)))  ;; not this clause
```

In none of the above cases does the clause selection consider the types of the objects within the list being destructured. Clause selection also based on type of object is a distinguishing feature of the `rte` based implementation of `destructuring-case`.

The `rte` type along with `destructuring-bind` and `type-case` as mentioned in Section 4.3.4 enables something similar to pattern matching in the XDuce language [HVP05]. The XDuce language allows the programmer to define a set of functions with various lambda lists, each of which serves as a pattern available to match particular target structure within an XML document. Which function gets executed depends on which lambda list matches the data found in the XML data structure.

XDuce introduces a concept called *regular expression types* which indeed seems very similar to *regular type expressions*. In [HVP05] Hosoya *et al.* introduce a *semantic type* approach to describe a system which enables their compiler to guarantee that an XML document conform to the intended type. The paper deals heavily with assuring that the regular expression types are well defined when defined recursively, and that decisions about subtype relationships can be calculated and exploited.

A notable distinction of the `rte` implementation as opposed to the XDuce language is that our proposal illustrates adding such type checking ability to an existing type system and suggests that such extensions might be feasible in other existing dynamic or reflective languages.

The concept of regular trees is more general than what `rte` supports, posing interesting questions regarding apparent shortcomings of our approach. The limitation that `rte` cannot be used to express trees of arbitrary depth as discussed in Section 4.4.2 seems to be a significant limitation of the Common Lisp type system. Furthermore, the use of `satisfies` in the `rte` type definition, seriously limits the `subtypep` function's ability to reason about the type. Consequently, programs cannot always use `subtypep` to decide whether two `rte` types are disjoint or equivalent, or even whether a particular `rte` type is empty. Neither can the compiler dependably use `subtypep` to make similar decisions to avoid redundant assertions in function declarations.

It is not clear whether Common Lisp could provide a mechanism whereby application programs which define new type specifiers via `deftype` might extend the behavior of `subtypep`. Having such a capability would allow such an extension for `rte`. Rational language theory does provide a well defined algorithm for deciding such questions, given the relevant rational expressions [HMU06, Sections 4.1.1, 4.2.1]. According to a private conversation with Pascal Costanza, there is some precedent in ContextL [CH05] for using anonymous classes created at run-time. These anonymous classes may be useful in coaxing `subtypep` into calculating subtype relations between `rte` based types. More research and experimentation are needed to determine whether this technique may be useful.

4.7 Conclusions and perspectives

In this chapter we presented a Common Lisp type definition, `rte`, which implements a declarative pattern-based approach for declaring types of heterogeneous sequences illustrating it with several motivating examples.

We further discussed the implementation of this type definition and its inspiration based in rational language theory. While the total computation needed for such type checking may be large, our approach allows most of the computation to be done at compile time, leaving only an $\mathcal{O}(n)$ complexity calculation remaining for run-time computation.

Our contributions are:

1. recognizing the possibility to use principles from rational theory to address the problem of dynamic type checking of sequences in Common Lisp,
2. adapting the Brzozowski derivative algorithm to sequences of Lisp types by providing an algorithm to symbolically decompose a set of Lisp types into an equivalent set of disjoint types,
3. implementing an efficient $\mathcal{O}(n)$ algorithm to pattern match an arbitrary Lisp sequence, and
4. implementing concrete `rte` based algorithms for recognizing certain commonly occurring sequence patterns.

The Common Lisp specification limits how much we can extend the type system. For example, it seems from the specification that a Common Lisp implementation is forbidden from allowing self-referential types, even in cases where it would be possible to do so. For future extensions to this research, we would like to experiment with extending the `subtypep` implementation to allow application level extensions, and therewith examine run-time performance when using `rte` based declarations within function definitions. As mentioned in Section 4.6, it may be possible to use anonymous classes or other features of the Metaobject Protocol [Pae93, KdRB91] to extend the behavior of `subtypep`. Similarly, we mentioned in Section 4.4.11, another use of the Metaobject Protocol and its dependent maintenance protocol, with which we detect and react to certain changes in the type lattice, primarily changes in CLOS classes during debugging (but potentially also while running deployed applications). It is unclear at this point, which kinds of such run-time changes in the type lattice can be detected. For example, `class-name` is a reader in ANSI CL, but an `setf`-able accessor Metaobject Protocol, which means that classes may be renamed at run-time. The impact on `rte` of such renaming is unknown. Can `rte` types be made to reference anonymous classes whose behavior might change at run-time? It is not known at this point how much such extension the Common Lisp specification and the MOP allow.

Another topic we would like to investigate is whether the core of this algorithm can be implemented in other dynamic languages, and to understand more precisely which features such a language would need to have to support such implementation.

For future extensions to this research we would like to experiment with extending the `subtypep` implementation to allow application level extensions, and therewith examine run-time performance when using `rte`-based declarations within function definitions.

Another topic we'd like to research is whether the core of this algorithm can be implemented in other dynamic languages, and to understand more precisely which features such a language needs to have to support such implementation.

Several open questions remain:

As Section 4.4.8 explains, there are some known bootstrapping issues involved in creating global functions as a side effect of `deftype` expansion. While Section 4.4.8 explains our ASDF extensions for working around these issues, the Google Common Lisp Style Guide [BF, Sec Pitfalls] suggests a different, and more robust solution. The solution is to use `asdf-finalizers:eval-at-toplevel` instead. We only discovered this ASDF feature late in our research and have not yet investigated whether it can be retro-fitted into the `rte` implementation.

Can regular type expressions be extended to implement more things we would expect from a regular expression library? For example, some regular expression libraries have a syntax for grouping subexpressions, and referring back to them later such as in `regexp-search-and-replace`? Additionally, would such a search and replace capability be useful?

Can this theory be extended to tackle unification? Can `rte` be extended to implement unification in a way which adds value?

One general problem in general with regular expressions is that if you use them to find whether a string (sequence in our case) does or does not match a pattern. We would often like to know why it fails to match. Questions such as "How far did it match?" or "Where did it fail to match?" would be nice to answer. It is currently unclear whether the `rte` implementation can at all be extended to support these features.

Part II

Binary Decision Diagrams

Figure 4.14: Me (center) with my brother and my cousin at Granny's house, eating watermelon. I was not yet interested in Boolean equations and apparently wasn't very interested in the watermelon.

The BDD data structure has countless applications to problems involving Boolean algebra. In the Art of Computer Programming [Knu09, Page iv], Donald Knuth writes, "[BDDs] have become the data structure of choice for Boolean functions and for families of sets, and the more I play with them the more I love them. For eighteen months I've been like a child with a new toy, being able now to solve problems that I never imagined would be tractable."

In Part II we recount a pedagogical development of Binary Decision Diagrams in Chapter 5, and we look in depth at questions of size of certain Binary Decision Diagrams in Chapter 6. In Chapter 7 we note that the subtype relation provides a challenge when representing the Boolean algebra of the Common Lisp type system using ROBDDs. In Chapter 7 we extend the ROBDDs to accommodate the Common Lisp type system.

Chapter 5

Reduced Ordered Binary Decision Diagrams

In Chapter 4 we described a procedure for recognizing heterogeneous sequences of objects according to regular type patterns. We noted that an initial implementation reveals several challenges of representation and efficient execution. The problems of efficient execution will be addressed in later chapters. In the current chapter we examine a data structure called the Binary Decision Diagram (BDD), which we will extend in Chapter 7 to accommodate this representational challenge.

The decision diagram has been defined in several different flavors in currently available literature. Colange [Col13, Section 2.3] provides a succinct historical perspective, including the BDD [Bry86], the Multi-Valued Decision Diagram (MDD) [Sri02], Interval Decision Diagram (IDD) [ST98], the Multi-Terminal Binary Decision Diagram (MTBDD) [CMZ+97], the Edge-Valued Decision Diagram (EVDD) [LS92], and the Zero-Suppressed Binary Decision Diagram (ZBDD) [Min93].

The particular decision diagram variant which we investigate in this report is the Reduced Ordered Binary Decision Diagram (ROBDD). When we use the term ROBDD we mean, as the name implies, that the BDD has its variables **O**rdered as described in Section 5.1.1 and has been fully **R**educed by the rules presented in Section 5.1.2. It is worth noting that there is variation in the terminology used by different authors. For example, Knuth [Knu09] and Bryant [Bry18] both use the unadorned term BDD for what we are calling an ROBDD.

Section 5.1 provides illustrations of ROBDD constructing from the point of view of reduction operations.

An equation of Boolean variables can be represented by a BDD [Bry86, Bry92, Ake78, FTV16] [Knu09, Section 7.1.4] [Col13, Section 2.3]. Andersen summarizes many of the algorithms for efficiently manipulating BDDs [And99]. Not least important in Andersen's discussion is how to use a hash table and dedicated constructor function to eliminate any redundancy within the same BDD or elsewhere within a forest of BDDs. Giuseppe Castagna [Cas16] introduces the connection of BDDs to type theoretical calculations, and provides straightforward algorithms for implementing set operations (intersection, union, relative complement) of types using BDDs.

Using the BDD data structure along with the algorithms described in [NVC17] we can efficiently represent and manipulate Common Lisp type specifiers. We may programmatically represent Common Lisp types largely independent of the actual type specifier representation. Moreover, unions, intersections, and relative complements of Common Lisp type specifiers can be calculated using the reduction BDD manipulation rules as well.

5.1 BDD reduction

We do not provide a formal definition of BDD here. Instead the interested reader is invited to consult any of the literature cited above. Instead we focus here on understaning implementation issues, with special emphasis on our BDD implementation in Common Lisp.

BDDs can be implemented easily in a variety of programming languages with only a few lines of code. The data structure provides a mechanism to manipulate Boolean expressions elegantly. Operations such as intersection, union and complement can be performed resulting in structures representing Boolean expressions in canonical form [Bry86]. The existence of this canonical form makes it possible to implement the equality predicate for Boolean expressions, either by straightforward structural comparison, or by pointer comparison depending on the specific BDD implementation. Some programming languages model types as sets [HVP05, CL17, Ans94]. In such programming languages, the BDD is a potentially useful tool for representing types and for performing certain type manipulations [Cas16, NVC17, NV18c].

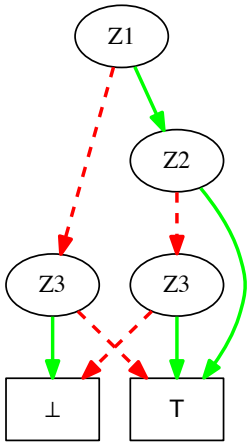


Figure 5.1: BDD for $(Z_1 \wedge Z_2) \vee (Z_1 \wedge \neg Z_2 \wedge Z_3) \vee (\neg Z_1 \wedge \neg Z_3)$

Figure 5.1 shows an example of a BDD which represents a particular function of three Boolean variables: Z_1, Z_2, and Z_3. The BDD in the figure is actually an ROBDD; we will define more precisely what that means later. When the Boolean function is expressed in Disjunctive Normal Form (DNF), it contains three terms, each of which corresponds to a path in the BDD from the root node, Z_1, to the leaf node, $\bot$. Paths from the root to the $\bot$ leaf must be ignored when serializing the DNF. The variables in each term are logically negated (*i.e.* $\neg Z_i$) if the path exits node Z_i via its dashed red exit arrow, and are not negated (*i.e.* Z_i) if the path follows the solid green exit arrow.

In order to avoid confusion, when this report is printed in black and white, we hereafter refer to the red dashed arrow as the negative arrow and the green solid arrow as the positive arrow. Respectively, we refer to the nodes which the arrows point to as the negative and positive children.

There are several conventions used in literature for graphically representing a Boolean expression as a BDD. Some conventions indicate the false (logically negated $\neg$) case as an arrow exiting the node on the bottom left and the true case as an arrow exiting the node on the left. We found that such a convention forces BDDs to be drawn with excessively many crossing lines. In order to allow the right/left arrows within the BDDs to be permuted, thus reducing the number of line crossings, we avoid attaching semantic information to left and right arrows, and instead use red dashed arrows for the false (negative) case and solid green arrows for the true (positive) case.

Casually generating a set of sample BDDs for randomly chosen Boolean expressions quickly reveals that a BDD may have redundant subtrees. It seems desirable to reduce the memory footprint of such a tree by reusing common subtrees (sharing pointers) or eliminating redundant nodes. Here, we introduce one such approach: first, we start with a complete binary tree (Section 5.1.1), and then, we transform it into a reduced graph by applying certain reduction rules to its nodes (Section 5.1.2). The process is intended to be intuitive, conveying an understanding of the topology of the resulting structure. On the contrary, this construction process is not to be construed as an algorithm for efficiently manipulating BDDs programmatically.

In addition to significantly reducing the memory footprint of a BDD, the optimization strategy described here also enables certain significant algorithmic optimizations, which we won't discuss in depth in this report. In particular, the equality of two Boolean expressions often boils down to a mere pointer comparison [NVC17].

5.1.1 Initial construction step

One way to understand the (RO)BDD representation of a Boolean expression is by first representing the truth table of the Boolean expression as decision diagram. Consider this Boolean expression of four variables: $\neg\ ((A \wedge C) \vee (B \wedge C) \vee (B \wedge D))$. Its truth table is given in Figure 5.3, and its BDD representation in Figure 5.2.

Definition 5.1. When a BDD is a tree corresponding exactly to the truth table of its Boolean function (which is not necessarily true), the BDD is referred to as an UOBDD, an unreduced ordered BDD.

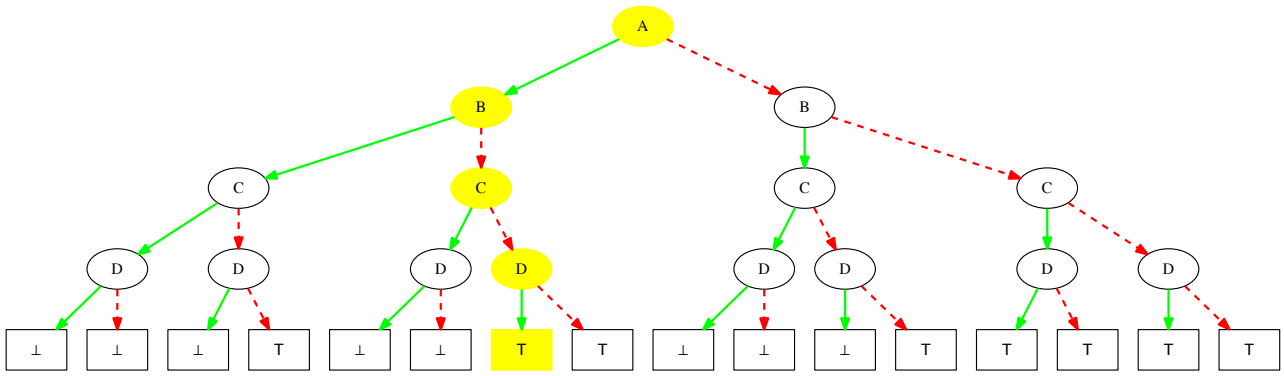


Figure 5.2: UOBDD for $F = \neg\ ((A \wedge C) \vee (B \wedge C) \vee (B \wedge D))$. The highlighted path of nodes corresponds to the highlighted row of the truth table in Figure 5.3.

A	B	C	D	F
⊤	⊤	⊤	⊤	⊥
⊤	⊤	⊤	⊥	⊥
⊤	⊤	⊥	⊤	⊥
⊤	⊤	⊥	⊥	⊤
⊤	⊥	⊤	⊤	⊥
⊤	⊥	⊤	⊥	⊥
⊤	⊥	⊥	⊤	⊤
⊤	⊥	⊥	⊥	⊤
⊥	⊤	⊤	⊤	⊥
⊥	⊤	⊤	⊥	⊥
⊥	⊤	⊥	⊤	⊥
⊥	⊤	⊥	⊥	⊤
⊥	⊥	⊤	⊤	⊤
⊥	⊥	⊤	⊥	⊤
⊥	⊥	⊥	⊤	⊤
⊥	⊥	⊥	⊥	⊤

Figure 5.3: Truth table for $F = \neg\ ((A \wedge C) \vee (B \wedge C) \vee (B \wedge D))$. The highlighted row of the truth table corresponds to the path of highlighted nodes in Figure 5.2.

Each non-leaf node of the UOBDD represents the appearance of a Boolean variable in the expression. Each path from the root node A to a leaf node represents one row of the truth table. For example, the highlighted path in Figure 5.2 corresponds to the highlighted row in Figure 5.3.

Definition 5.2. A BDD is said to be ordered if there is some ordering of the variables $\{v_1, v_2, ..., v_n\}$, such that whenever there is an arrow from v_i to v_j then $i < j$.

Some authors such as Gröpl *et al.* [GPS98, GPS01] and Langberg *et al.* [LPR03], when exploring a BDD variant called qOBDD, consider the further restriction that arrows only connect v_i with v_{i+1}. We do not make such a restriction in our treatment of ROBDDs. For UOBDDs having not yet undergone any reduction, being ordered implies that every branch from the root to a leaf contains exactly the same variables in the same order. In Figure 5.2 for example, every path from the root to a leaf contains exactly A, B, C, D in that order. On the contrary, and as in Figure 5.1, some of these paths contain fewer nodes than others. Nevertheless, the nodes visited by each path remain ordered. For the extent of this paper we will use the natural lexicographical orderings: $A < B < C < D$, and $Z_1 < Z_2 < ... < Z_n$.

5.1.2 Reduction rules

The diagram in Figure 5.2 is an ordered BDD, but not an ROBDD, because the BDD has not yet been reduced. Systematic application of the three reduction rules described in this section will convert the ordered BDD into an ROBDD.

Given its UOBDD, it is straightforward to evaluate an arbitrarily complex Boolean expression of n variables, simply descend the tree in n steps according to the values of the n variables. This is equivalent to tracing across the corresponding row of the truth table (see the highlighting in Figure 5.3). However, the size of the tree grows exponentially with the number of variables. The UOBDD representing a Boolean expression of n variables has $UOBDD$ number of nodes as indicated in Equation 5.1.

Notation 5.3.

$$|UOBDD_n| = 2^{n+1} - 1 \tag{5.1}$$

Fortunately, it is possible to reduce the allocated size of the UOBDD by taking advantage of certain redundancies. There are three rules which can be used to guide the reduction. Andersen and Gröpl [And99, GPS98] also explain the merging and deletion rules, so we will dispense with many of the details. However, we consider an extra rule, the terminal rule, which is really a special case of the merging rule.

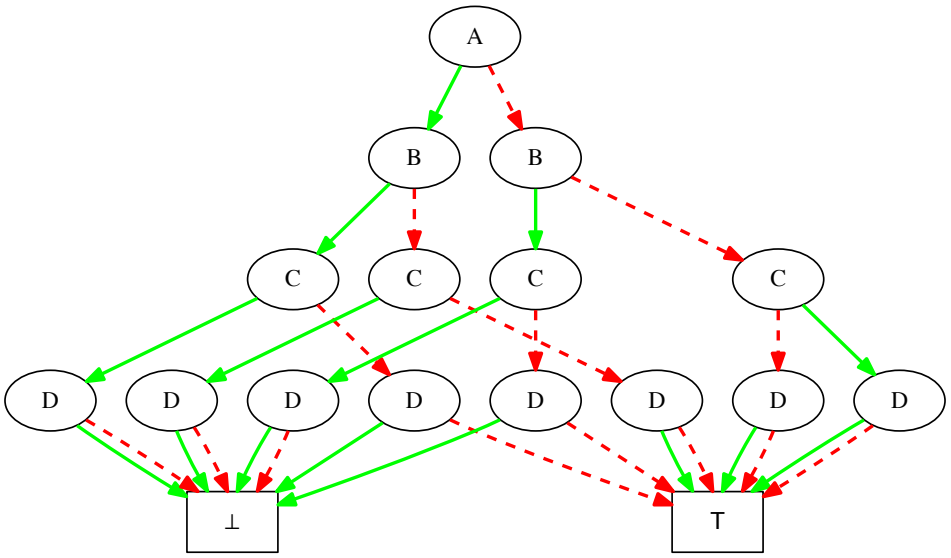


Figure 5.4: BDD after applying Terminal rule

Definition 5.4. If any node X in an ROBDD is such that its positive and negative children are the same node Y, then X is said to be symmetric.

Definition 5.5. If any two nodes U and V corresponding to the same Boolean variable are such that their positive children are both node X and negative children are both node Y, then U and V are said to be congruent.

Terminal rule: The only possible leaf nodes are $\top$ and $\bot$, so these nodes can be represented by two singleton objects, allowing pointers to them to be shared.

Deletion rule: If node X is symmetric (Definition 5.4) and has Y as a child, node X can be deleted and arrows previously pointing to it may be promoted to point to Y directly.

Merging rule: If nodes U and V are congruent (Definition 5.5) they may be merged. Any arrow pointing to U may be updated to point to the V, and U may be removed (or the other way around).

Applying the Terminal rule reduction cuts the number of nodes roughly by half, as shown in Figure 5.4.

Figure 5.6 illustrates reductions in the graph as a result of applying the deletion rule multiple times. In this case the graph shrinks from 17 nodes to 10 nodes after two applications of the rule.

Figure 5.7 illustrates reductions in the graph as a result of applying the merging rule multiple times. In the figure the two nodes marked D are congruent, and can thus be reduced to a single D node. Thereafter, the two highlighted C nodes are seen to be congruent, and can thus be merged. In this case the graph shrinks from 10 nodes to 8 nodes after two applications of the rule.

Application of the three rules results in the ROBDD shown in Figure 5.5. In this case the graph shrinks from 31 nodes in Figure 5.2 to 8 nodes in Figure 5.5.

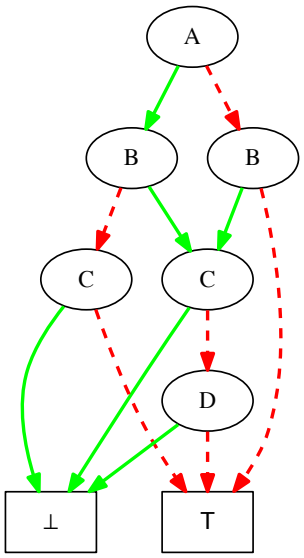
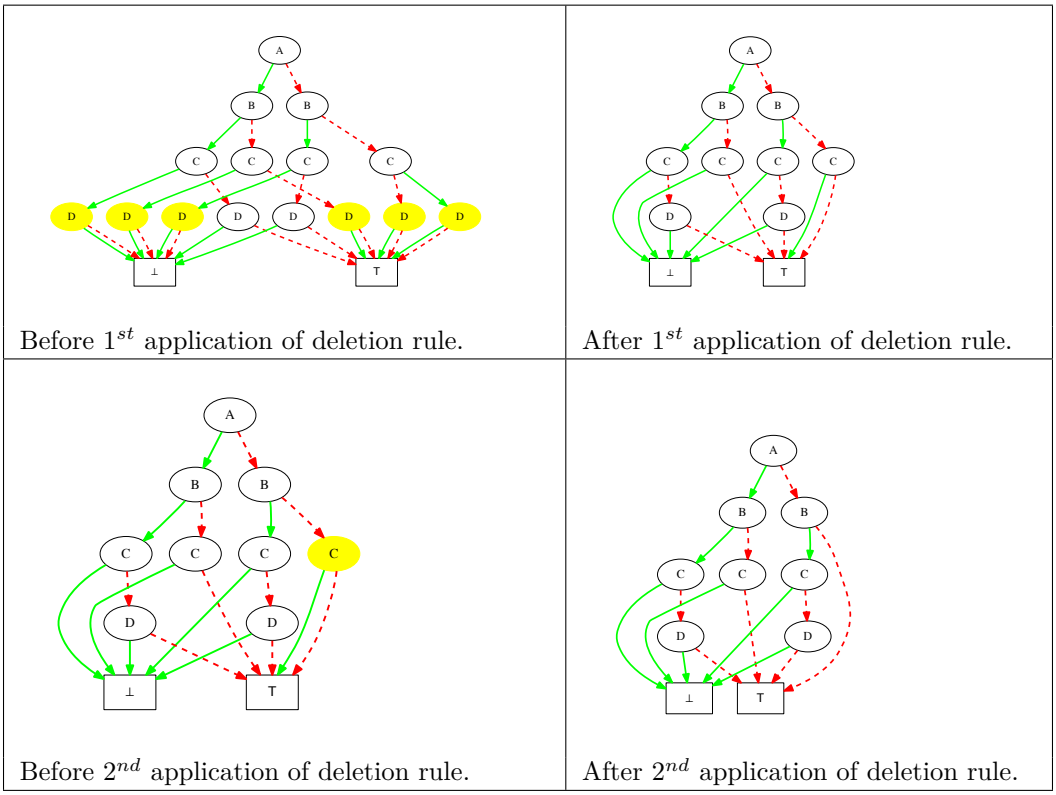


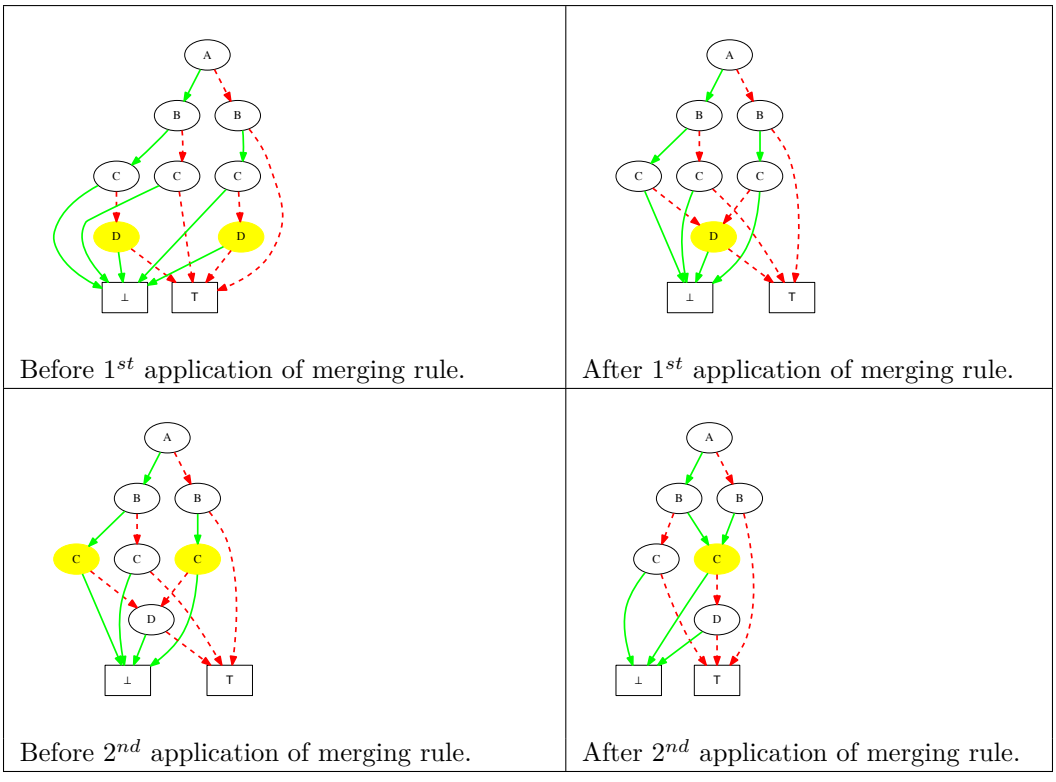
Figure 5.5: BDD after applying Terminal, Deletion, and Merging rules. This diagram is an ROBDD logically equivalent to the UOBDD shown in Figure 5.2.



Before 1^{st} application of deletion rule. | After 1^{st} application of deletion rule.

Before 2^{nd} application of deletion rule. | After 2^{nd} application of deletion rule.

Figure 5.6: Examples of Deletion Rule. The highlighted nodes are symmetric. Such nodes are deleted.



Before 1^{st} application of merging rule. | After 1^{st} application of merging rule.

Before 2^{nd} application of merging rule. | After 2^{nd} application of merging rule.

Figure 5.7: Examples of Merging Rule. The highlighted nodes in the left hand-column are congruent. One is deleted, one remains, as shown in the right-hand column.

5.2 ROBDD Boolean operations

Notation 5.6. In an abstract Boolean algebraic context

$$A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)\,.$$

In a set theoretical context

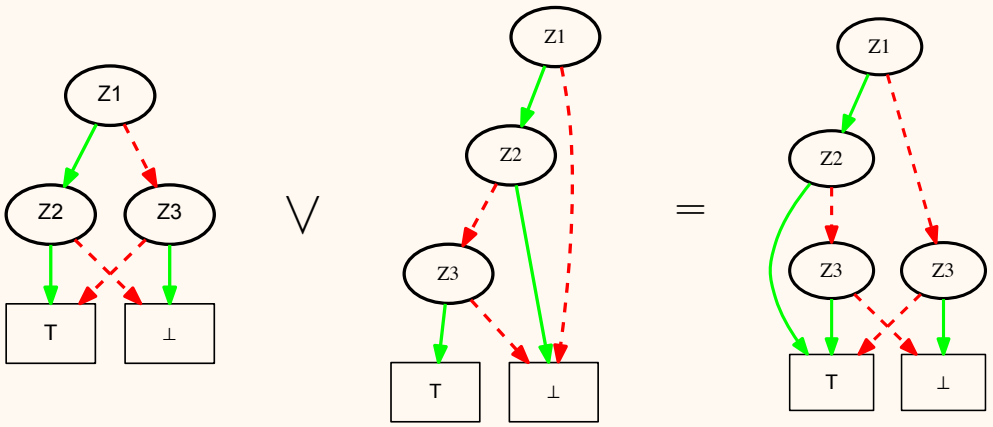
$$A \oplus B = (A \backslash B) \cup (B \backslash A)\,.$$

In this section we discuss the algorithms for computing the ROBDD representing the Boolean combination of two given ROBDDs. In Equation (5.2) we state the concise formula in terms of the components of the two given ROBDDS. Equation (5.2) holds for the commutative operations of conjunction ($\wedge$), disjunction ($\vee$), symmetric difference which may also be referred to as exclusive-or ($\oplus$), and also for the non-commutative relative complement ($\backslash$) operation.

As an illustration, Example 5.7 shows the Boolean-or of the two ROBDDs representing $(Z_1 \wedge Z_2 \vee \neg Z_1 \wedge \neg Z_2)$ and $(Z_1 \wedge \neg Z_2 \wedge Z_3)$. And Example 5.8 shows the Boolean-not operation applied to the ROBDD representing $Z_1 \wedge Z_2 \vee \neg Z_1 \wedge \neg Z_2 \vee Z_1 \wedge \neg Z_2 \wedge Z_3$.

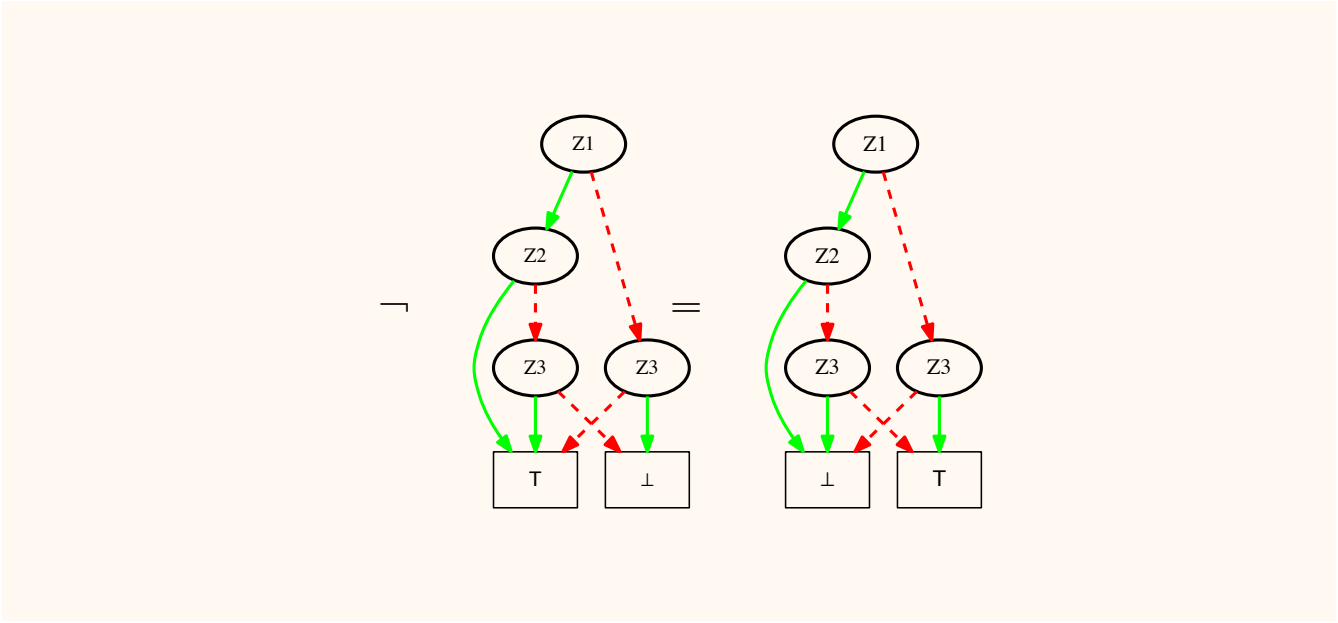
Example 5.7 (Boolean-or of two ROBDDs)**.**

$$(Z_1 \wedge Z_2 \vee \neg Z_1 \wedge \neg Z_2) \vee (Z_1 \wedge \neg Z_2 \wedge Z_3) \qquad = Z_1 \wedge Z_2 \vee \neg Z_1 \wedge \neg Z_2 \vee Z_1 \wedge \neg Z_2 \wedge Z_3$$



Example 5.8 (Boolean-not of an ROBDD effectively swaps the terminal nodes.)**.**

$$\neg(Z_1 \wedge Z_2 \vee \neg Z_1 \wedge \neg Z_2 \vee Z_1 \wedge \neg Z_2 \wedge Z_3) \qquad = Z_1 \wedge \neg Z_2 \wedge \neg Z_3 \vee \neg Z_1 \wedge Z_3$$



The general recursive algorithms for computing the BDDs which represent the common Boolean algebra operators are straightforward.

$$
\begin{aligned}
\langle\text{robdd-node}\rangle &\models \langle\text{internal-node}\rangle \mid \langle\text{terminal-node}\rangle \\
\langle\text{internal-node}\rangle &\models \text{Node}(\langle\text{var}\rangle\,\langle\text{robdd-node}\rangle\,\langle\text{robdd-node}\rangle) \\
\langle\text{var}\rangle &\models \textit{any variable name label} \\
\langle\text{terminal-node}\rangle &\models \bot \mid \top
\end{aligned}
$$

Figure 5.8: Notation for ROBDDs including Node constructor syntax.

Let B, B_1, B_2, C_1, C_2, D_1, and D_2 denote ROBDDs, with $B_1 = Node(v_1\ C_1\ D_1)$ and $B_2 = Node(v_2\ C_2\ D_2)$, and let v_1 and v_2 denote variable names. For the following Equations (5.2) and (5.3) to be valid, it is necessary that v_1 and v_2 represent variable names that are order-able. We would eventually like the labels to accommodate Common Lisp type specifiers, but this is not immediately possible. This extension to Common Lisp types is discussed in Chapter 7 in Section 7.3.

Castagna [Cas16] presents Equations (5.2) and (5.3) in a form which is slightly different syntactically. The formulas for $(B_1 \vee B_2)$, $(B_1 \wedge B_2)$, $(B_1 \oplus B_2)$, and $(B_1 \setminus B_2)$ are similar to each other. If $\circ \in \{\vee, \wedge, \oplus, \setminus\}$, then

$$
B_1 \circ B_2 = \begin{cases} Node(v_1 & (C_1 \circ C_2) & (D_1 \circ D_2)) & \text{for } v_1 = v_2 \\ Node(v_1 & (C_1 \circ B_2) & (D_1 \circ B_2)) & \text{for } v_1 < v_2 \\ Node(v_2 & (B_1 \circ C_2) & (B_1 \circ D_2)) & \text{for } v_1 > v_2 \end{cases} \tag{5.2}
$$

$$
\neg B = \top \setminus B \tag{5.3}
$$

As $\top$ or $\bot$ do not have children, there are several special cases which apply if either $\top$ or $\bot$ appear as one of the operands of the operation. Another special case is when $B_1 = B_2$, no recursive call need be made. Figure 5.9 details these special cases. The first three of these rules serve as termination conditions for the recursive algorithms.

$\wedge$ Operations	$\vee$ Operations	$\oplus$ Operations	$\backslash$ Operations	Reduce to
	$\top \vee B$ $B \vee \top$			$\top$
$\bot \wedge B$ $B \wedge \bot$		$B \oplus B$	$B \backslash \top$	$\bot$
$B \wedge B$ $\top \wedge B$ $B \wedge \top$	$B \vee B$ $\bot \vee B$ $B \vee \bot$	$\bot \oplus B$ $B \oplus \bot$	$B \backslash \bot$	B
		$B \oplus \top$ $\top \oplus B$		$\top \backslash B$
			$\top \backslash Node(B\ B_1\ B_2)$	$Node(B\ (\top \backslash B_1)\ (\top \backslash B_2))$

Figure 5.9: Special cases for ROBDD Boolean operations. When either argument is $\top$ or $\bot$ or both arguments are the same. Note that $\wedge$, $\vee$, and $\oplus$ are commutative operations, but $\backslash$ is not.

We include exclusive-or even though it is not normally a fundamental operation. One might alternatively compute $\oplus$ by either of the following identities:

$$B_1 \oplus B_2 = (B_1 \backslash B_2) \vee (B_2 \backslash B_1) \tag{5.4}$$
$$B_1 \oplus B_2 = (B_1 \vee B_2) \backslash (B_1 \wedge B_2)\,. \tag{5.5}$$

Such a definition would indeed be semantically correct, although the number of operations in such a computation would often be higher than if (5.2) were used.

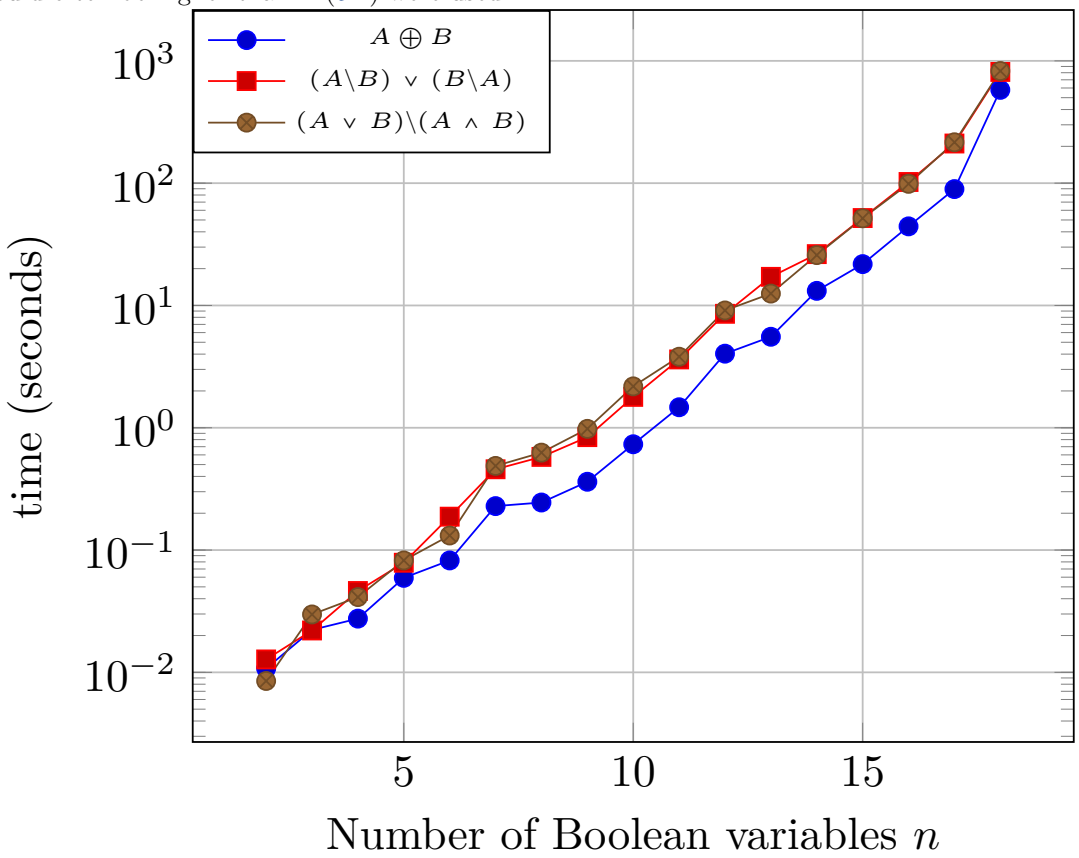


Figure 5.10: Comparison of execution time of three proposed implementations of the ROBDD xor, $\oplus$ operator. The plot shows that calculating $\oplus$ is usually faster using Equation (5.2) as opposed to Equation (5.4) or (5.5)

We performed experiments to compare the run-time performances of the computation of Equations (5.2), (5.4), and (5.5). Sections 5.3.1, 5.4, and 5.5 show Common Lisp implementations of the `bdd` class and functions to manipulate its objects, including an implementation of the Boolean functions. Figure 5.10 shows a plot of computation times of the three equations applied to randomly generated ROBDDs. As an example on a MacBook Air^{TM} with 1.8 GHz Intel Core i5 processor and 8 GB 1600 MHz DDR3 memory, calculation of exclusive-or of two 12-variable ROBDDs took an average of 3.5ms using the algorithm from Equation (5.2) compared to 8.8ms using $(A \backslash B) \vee (B \backslash A)$ and 7.9ms using $(A \vee B) \backslash (A \wedge B)$.

5.3 ROBDD construction

Section 5.1.1 may serve as a pedagogical explanation of how to construct an ROBDD. However, a much more efficient mechanism, a memory function, was presented by Bryant [Bry86]. Brace *et al.* [BRB90] made this procedure explicit in what they called the ite (if-then-else) algorithm. Finally Andersen [And99] presented an easy to follow set of guidelines. Algorithm 2 is based closely on Andersen's presentation.

Algorithm 2: BddNode Efficient ROBDD construction

Input: var Boolean variable
Input: $node_{pos}$ Positive child node
Input: $node_{neg}$ Negative child node
Input: H a mutable mapping of $(variable, node, node) \rightarrow node$
Output: A node representing the constructed node

2.1 **if** $node_{pos} = node_{neg}$ **then**
2.2 | **return** $node_{pos}$ // enforce deletion rule
2.3 **else if** lookup $H[var, node_{pos}, node_{neg}]$ **then**
2.4 | **return** $H[var, node_{pos}, node_{neg}]$ // enforce merging rule
2.5 **else**
2.6 | $node_{new} \leftarrow NewNode(var, node_{pos}, node_{neg})$
2.7 | $H[var, node_{pos}, node_{neg}] \leftarrow node_{new}$
2.8 | **return** n_{new}

The management of the mapping, H, in Algorithm 2 deserves special discussion. In our implementation, in Common Lisp, we chose to use a dynamically scoped hash table. An alternative approach which we did not investigate would have been to store the hash table as a slot in the `bdd` object.

One problem which arises at run-time, especially during prolonged calculations is that this hash table may become large. Its purpose, according to Bryant and Brace, is to efficiently reduce needless reallocation, and by doing so, to enforce the merging rule as indicated on line 2.4 of Algorithm 2. However, as is discussed in Chapter 6, the space of all possible ROBDDs, and thus of all possible ROBDD nodes, is many orders of magnitude larger than the number of nodes actually allocated in a program. The resulting risk of this massive size difference is that a program may create a large number of entries in the hash table, but rarely look them up again.

The problem of ever growing hash table can be mitigated somewhat. We have implemented several strategies to address this problem. They are explained in Section 5.3.3. The performance differences of the strategies is discussed in Section 10.5 and illustrated in Figure 10.3.

5.3.1 Common Lisp ROBDD implementation

Implementation 5.9 shows the Clos definition of the base class `bdd` which is used as the parent class for terminal nodes and internal nodes defined respectively in Implementations 5.10 and 5.11. The class, `bdd` contains a slot named `ident` which is a positive integer uniquely identifying each node. We exploit this slot and in a print-object method used in debugging. This slot would not be absolutely necessary if we were attempting to minimize the memory footprint.

Implementation 5.9 (CLOS definition of `bdd` *base* class)**.**

```
(defvar *bdd-node-count* 1)
(defclass bdd ()
  ((ident :reader bdd-ident
          :type unsigned-byte
          :initarg :ident
          :initform (incf *bdd-node-count*))
   (label :reader bdd-label
          :initarg :label)))
```

Implementation 5.10 shows the class definitions and singleton object definitions of `*bdd-true*` and `*bdd-false*`. All applications using these classes are required to voluntarily use `*bdd-true*` and `*bdd-false*` rather than allocating new objects; the terminal rule (Section 5.1.2) will be implicitly enforced. We forego any

further discussion of techniques for enforcing singleton classes in CLOS and refer the reader to the discussion by Marchand *et al.* [Mar98].

Implementation 5.10 (CLOS definition of `bdd` leaf/terminal nodes)**.**

```
(defclass bdd-leaf (bdd) ())

(defclass bdd-true (bdd-leaf)
  ((ident :initform 1)
   (label :initform t)))

(defclass bdd-false (bdd-leaf)
  ((ident :initform 0)
   (label :initform nil)))

(defvar *bdd-true* (make-instance 'bdd-true))
(defvar *bdd-false* (make-instance 'bdd-false))
```

Implementation 5.11 is the class definition for internal nodes. Each such nodes has a *positive* and *negative* child, each of which may be another internal node, or a leaf node, so we have included a type declaration for these slots: `(or bdd-node bdd-leaf)`. Notice that a leaf node, `bdd-leaf`, has no children.

Implementation 5.11 (CLOS definition of `bdd-node` class, used for internal non-terminal nodes)**.**

```
(defclass bdd-node (bdd)
  ((positive :type (or bdd-node bdd-leaf)
             :initarg :positive
             :reader bdd-positive)
   (negative :type (or bdd-node bdd-leaf)
             :initarg :negative
             :reader bdd-negative)))

(defmethod print-object ((bdd bdd-node) stream)
  (print-unreadable-object (bdd stream :type nil :identity nil)
    (format stream "bdd[~D]->" (bdd-ident bdd))
    (format stream "->[+~S,-~S]" (bdd-ident (bdd-positive bdd))
                                 (bdd-ident (bdd-negative bdd)))))
```

Implementation 5.20 calls a function named `bdd-node`. We show the definition of this function in Implementation 5.12. This implementation follows the model presented in Algorithm 2.

Implementation 5.12 (`bdd-node`: Retrieve or allocate a new `bdd-node` object with the given variable name and children.)**.**

```
(defmethod bdd-node (var-name (positive-bdd bdd) (negative-bdd bdd))
  (check-type *bdd-hash* 'hash-table
     "*bdd-hash* must be dynamically bound to a hash-table")
  (cond
    ((eq positive-bdd negative-bdd)
     positive-bdd)
    ((gethash *bdd-hash* (list var-name positive-bdd negative-bdd)))
    (t
     (setf (gethash *bdd-hash* (list var-name positive-bdd negative-bdd))
           (make-instance 'bdd-node :label var-name
                                    :positive positive-bdd
                                    :negative negative-bdd)))))
```

The function `bdd-node` should only be called from within the dynamic extent of `bdd-with-new-hash` as shown in Example 5.13. The example shows the calculation of the following Boolean expression:

$$(x_1 \wedge \neg x_2) \vee x_2 \,.$$

Example 5.13 (Call to `bdd-node` within `bdd-call-with-new-hash`.)**.**

```
(bdd-with-new-hash
  (bdd-or
    (bdd-and (bdd-node 'x1 *bdd-true* *bdd-false*)
             (bdd-not (bdd-node 'x2 *bdd-true* *bdd-false*)))
    (bdd-node 'x2 *bdd-true* *bdd-false*)))
```

The code in Example 5.13 references the functions `bdd-or`, `bdd-and`, and `bdd-not` which are discussed in Section 5.4; a discussion of `bdd-with-new-hash` can be found in Section 5.3.3.

5.3.2 BDD object serialization and deserialization

While Example 5.13 is semantically correct, we would much rather be able to specify the `bdd` object representing $(x_1 \wedge \neg x_2) \vee x_2$ via an s-expression such as `(or (and x1 (not x2)) x2)`.

In order to serialize an ROBDD into a Boolean expression in DNF (disjunctive normal form) we can simply walk the `bdd` object from root to leaf (traversing every such path). Each time `*bdd-true*` is reached, collect the path, with appropriate variable inversions. The function `bdd-to-dnf` in Implementation 5.14 shows one such procedure. The function could of course be further optimized to eliminate singleton `and`/`or` expressions, so as to generate `(OR X1 X1)` rather than `(OR (AND X1) (AND X2))`.

Implementation 5.14 (`bdd-to-dnf`: Serialize a `bdd` object into an s-expression denoting the DNF of the Boolean expression.)**.**

```
(defun bdd-to-dnf (bdd)
  (let (or-terms)
    (labels ((partial-dnf (node and-term)
               (typecase node
                 (bdd-true
                    (push (cons 'and (reverse and-term)) or-terms))
                 (bdd-false nil)
                 (bdd-node
                    (bdd-label node)
                    (partial-dnf (bdd-positive node)
                                 (cons (bdd-label node) and-term))
                    (partial-dnf (bdd-negative node)
                                 (cons `(not ,(bdd-label node)) and-term))))))
      (partial-dnf bdd nil)
      (cons 'or or-terms))))
```

We see how `bdd-to-dnf` is used in Example 5.15. In the example, we see that the Boolean expression: $(x_1 \wedge \neg x_2) \vee x_2$ is printed as the s-expression `(OR (AND (NOT X1) X2) (AND X1))` which is equivalent to the Boolean expression $(\neg x_1 \wedge x_2) \vee \neg x_1$. The reader can easily verify that

$$(\neg x_1 \wedge x_2) \vee \neg x_1 = (x_1 \wedge \neg x_2) \vee x_2 \,.$$

Example 5.15 (Example call to `bdd-to-dnf`.)**.**

```
CL-USER> (bdd-with-new-hash
           (bdd-to-dnf
             (bdd-or (bdd-and (bdd-node 'x1 *bdd-true* *bdd-false*)
                              (bdd-not (bdd-node 'x2 *bdd-true* *bdd-false*)))
                     (bdd-node 'x2 *bdd-true* *bdd-false*))))

(OR (AND (NOT X1) X2) (AND X1))
```

The function which takes an s-expression representing a Boolean expression is simply called `bdd` rather than `bdd-dnf-to-bdd`, because it is capable of converting any valid such Boolean expression, not limited to DNF. We include the operators `and-not` and `xor` which are discussed in Sections 5.4 and 5.20.

Implementation 5.16 (`bdd`: convert any Boolean expression into a `bdd` object.)**.**

```
(defun bdd (expr)
  (typecase expr
    ((eql t)
     *bdd-true*)
    ((eql nil)
     *bdd-false*)
    (symbol
     (bdd-node expr *bdd-true* *bdd-false*))
    ((not list)
     (error "invalid expression ~A" expr))
    (t
     (apply
       (case (car expr)
         ((not)     #'bdd-not)
         ((and)     #'bdd-and-list)
         ((or)      #'bdd-or-list)
         ((and-not) #'bdd-and-not-list)
         ((xor)     #'bdd-xor-list)
         (t (error "invalid expression ~A" expr)))
       (mapcar #'bdd (cdr expr))))))
```

Using the `bdd` function, shown in Implementation 5.16, we can simplify the expression from Example 5.15. The reformulated call is shown in Example 5.17.

Example 5.17 (Simplified version of Example 5.15, and other Boolean expressions.)**.**

```
CL-USER> (bdd-with-new-hash
           (bdd-to-dnf (bdd '(or (and x1 (not x2))
                                 x2))))

(OR (AND (NOT X1) X2)
    (AND X1))

CL-USER> (bdd-with-new-hash
           (bdd-to-dnf (bdd '(xor (or (and x1 (not x2))
                                      (and-not x2 (or (not x2)
                                                      x1)))))))

(OR (AND X1 (NOT X2))
    (AND (NOT X1) X2))
```

5.3.3 BDD retrieval via hash table

Section 5.3 introduced a commonly used technique of maintaining a mapping (in our case, a hash table) to avoid multiple allocation of BDDs. Additionally we included an assertion for the non-vacuity of `*bdd-hash*` in the method definition of `bdd-node` in Implementation 5.12. Implementation 5.18 defines the macro `bdd-with-new-hash`. All manipulations of `bdd` objects must be performed within the dynamic extent of a call to `bdd-call-with-new-hash` which this macro enforces.

Implementation 5.18 (`bdd-with-new-hash`: Provide dynamic extent to allocate `bdd` objects.)**.**

```
(defvar *bdd-hash* nil "The default value of *bdd-hash* is NOT a hash table")

(defun bdd-call-with-new-hash (thunk)
  (let ((*bdd-hash* (make-hash-table :test #'equal)))
    (funcall thunk)))

(defmacro bdd-with-new-hash (&body body)
   `(bdd-call-with-new-hash (lambda () ,@body)))
```

The code in Implementation 5.18 is shown because it is easy to understand, but there are several ways to optimize this code, which have a noticeable effect on performance. Chapter 10 deals with several different performance issues, and Section 10.4 presents some alternative implementations of `bdd-call-with-new-hash`.

5.4 Common Lisp implementation of ROBDD Boolean operations

In this section we show a high-level overview of the Common Lisp implementation of the ROBDD Boolean operators whose Common Lisp names are: `bdd-or`, `bdd-and`, `bdd-xor`, and `bdd-and-not`. The binary functions, shown in Implementation 5.19, are implemented according to the same pattern, as a conditional with two branches. The consequence branch of each conditional handles the case of equal arguments (as indicated in Figure 5.9). The alternative branch of each condition is a call to `bdd-op` which implements Equation (5.2). The Common Lisp code for `bdd-op` can be found in Implementation 5.20.

Implementation 5.19 (ROBDD Boolean operations)**.**

```
(defmethod bdd-or ((b1 bdd-node) (b2 bdd-node))
  (if (eq b1 b2)
      b1
      (bdd-op #'bdd-or b1 b2)))

(defmethod bdd-xor ((b1 bdd-node) (b2 bdd-node))
  (if (eq b1 b2)
      *bdd-false*
      (bdd-op #'bdd-xor b1 b2)))

(defmethod bdd-and ((b1 bdd-node) (b2 bdd-node))
  (if (eq b1 b2)
      b1
      (bdd-op #'bdd-and b1 b2)))

(defmethod bdd-and-not ((b1 bdd-node) (b2 bdd-node))
  (if (eq b1 b2)
      *bdd-false*
      (bdd-op #'bdd-and-not b1 b2)))

(defmethod bdd-not ((b bdd-node))
  (bdd-and-not *bdd-true* b))
```

The code in Implementation 5.20 makes a call to the function `bdd-cmp` whose responsibility it is to compare the labels on the two given nodes. Recall that an ROBDD has ordered variables, which means that `bdd-cmp` returns a consistent value which is transitive in the sense that

$$v_1 < v_2 \text{ and } v_2 < v_3 \text{ implies } v_1 < v_3 \,.$$

In `bdd-op`, if the labels (Boolean variable names) are found to be equal, then the same operation is called recursively twice, once on the two positive branches, and once on the two negative branches. The function `bdd-node` is called with the values returned from the two recursive call, along with the common label. It is the job of `bdd-node` to either allocate a new internal node, or to retrieve a node from the cache if one already exists with the given label, positive child, negative child.

In the case that labels are found not to be equal, `bdd-op` performs the recursive call twice either with the same first argument or with the same second argument depending on whether the two given nodes were found to be in increasing or decreasing order, thereafter making a single call to `bdd-node` with the lesser of the two labels, and the two new children generated by the recursive calls. The `bdd-op` function takes care never to reverse the order of the arguments to the recursive call, as the given operation may not be commutative, as in fact the `bdd-and-not` operation is not commutative. See Implementation 5.20 for the exact details.

Implementation 5.20 (ROBDD Generic Boolean operation)**.**

```
(defun bdd-op (op bdd-1 bdd-2)
  (let ((lab-1      (bdd-label bdd-1))
        (positive-1 (bdd-positive bdd-1))
        (negative-1 (bdd-negative bdd-1))
        (lab-2      (bdd-label bdd-2))
        (positive-2 (bdd-positive bdd-2))
        (negative-2 (bdd-negative bdd-2)))
    (ecase (bdd-cmp lab-1 lab-2)
      ((=)
       (bdd-node lab-1 (funcall op positive-1 positive-2)
                       (funcall op negative-1 negative-2)))
      ((<)
       (bdd-node lab-1 (funcall op positive-1 bdd-2)
                       (funcall op negative-1 bdd-2)))
      ((>)
       (bdd-node lab-2 (funcall op bdd-1 positive-2)
                       (funcall op bdd-1 negative-2))))))
```

5.5 ROBDD operations with multiple arguments

Section 5.4 explains the implementation of the ROBDD Boolean binary operations. It is a fairly common situation to want to calculate these Boolean operations with multiple arguments. A common such case occurs when constructing an ROBDD from a sum-of-products formula, such as the one shown in Example 5.21.

Example 5.21 (Boolean expression in the form of sum of products)**.**

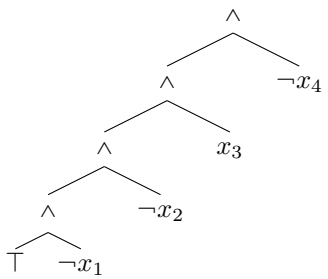
$$\begin{aligned} F = {} & x_1 \wedge x_2 \wedge x_3 \wedge x_4 \\ \vee \quad & \neg x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4 \\ \vee \quad & x_1 \wedge x_2 \wedge x_3 \wedge x_4 \\ \vee \quad & \neg x_1 \wedge x_2 \wedge \neg x_3 \wedge x_4 \\ \vee \quad & x_2 \wedge \neg x_3 \wedge x_4 \end{aligned}$$

A natural approach to calculating the ROBDD corresponding to Example 5.21 is with a fold operation which is a common higher-order function in many functional languages [Hut99]. There are two flavors of fold which we will examine: linear fold, and tree-like fold.

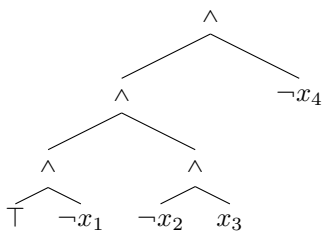
We examine the two flavors of fold here because we found the result surprising. We were not able to explain the results. We had some anecdotal evidence from previous experiments that tree-like fold would be faster when performing multiple-argument ROBDD Boolean operations. However, on constructing the explicit tests detailed in this section, we were unable to duplicate our previous results. We present the explanation here and invite others to falsify out results.

Consider the second term in Example 5.21, $\neg x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4$. Since the $\wedge$ operation is associative, this expression may be viewed in two different ways. Equation 5.6 represents the diagram of the linear fold computation, while Equation 5.7 represents the tree-like fold computation.

$$\neg x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_4 = \quad \text{linear fold} \tag{5.6}$$



$$= \quad \text{tree-like fold} \tag{5.7}$$



The Common Lisp `reduce` function [Ste90, Chapter 14.2] implements a linear fold, and we have provided the tree-like fold in Implementation 5.22.

Implementation 5.22 (`bdd-reduce`)**.**

```
(defun tree-fold (op bdd-list unit)
  (labels ((compactify (stack)
             (if (null (cdr stack))
                 stack
                 (destructuring-bind ((depth1 bdd1) (depth2 bdd2) &rest tail) stack
                   (if (= depth1 depth2)
                       (compactify (cons (list (1+ depth1) (funcall op bdd1 bdd2))
                                         tail))
                       stack))))
           (finish-stack (acc stack)
             (if (null stack)
                 acc
                 (finish-stack (funcall op acc (cadr (car stack)))
                               (cdr stack)))))
    (destructuring-bind ((depth bdd) &rest tail)
        (reduce (lambda (stack bdd)
                  (compactify (cons (list 1 bdd) stack)))
                bdd-list
                :initial-value (list (list 1 unit)))
      (finish-stack bdd tail))))
```

Definition 5.23. If a Boolean function of n is expressed as DNF, *i.e.* as a sum of m products, denote the number of variables used in the i'th product as ℓ_i. The density of the formula is

$$\eta = \frac{1}{m} \sum_{i=1}^{m} \ell_i \,.$$

I.e., η measures the average number of variables per DNF term.

The density is a function only of the representation of the Boolean function, not of the Boolean function itself. There may well be many different formulas for the same Boolean function with different densities.

A sum of minterms has $\eta = 100\%$ as every minterm contains all the Boolean variables either complemented ($\neg x_i$) or non-complemented.

While these two forms of folding are semantically the same for any given associative binary function, it turns out that in terms of computation, they may be different. Figure 5.11 compares the construction times using both approaches. It is not surprising that as η decreases, the computation time decreases. We also observe from the plots in Figure 5.11, that the linear fold (based on the built-in Common Lisp `reduce` function) performs marginally better than our implementation of `tree-fold`, and in the case of $\eta = 22\%$ the linear fold performs much better for $n > 10$. This is curious, and somewhat counter-intuitive. More research is needed to explain this result.

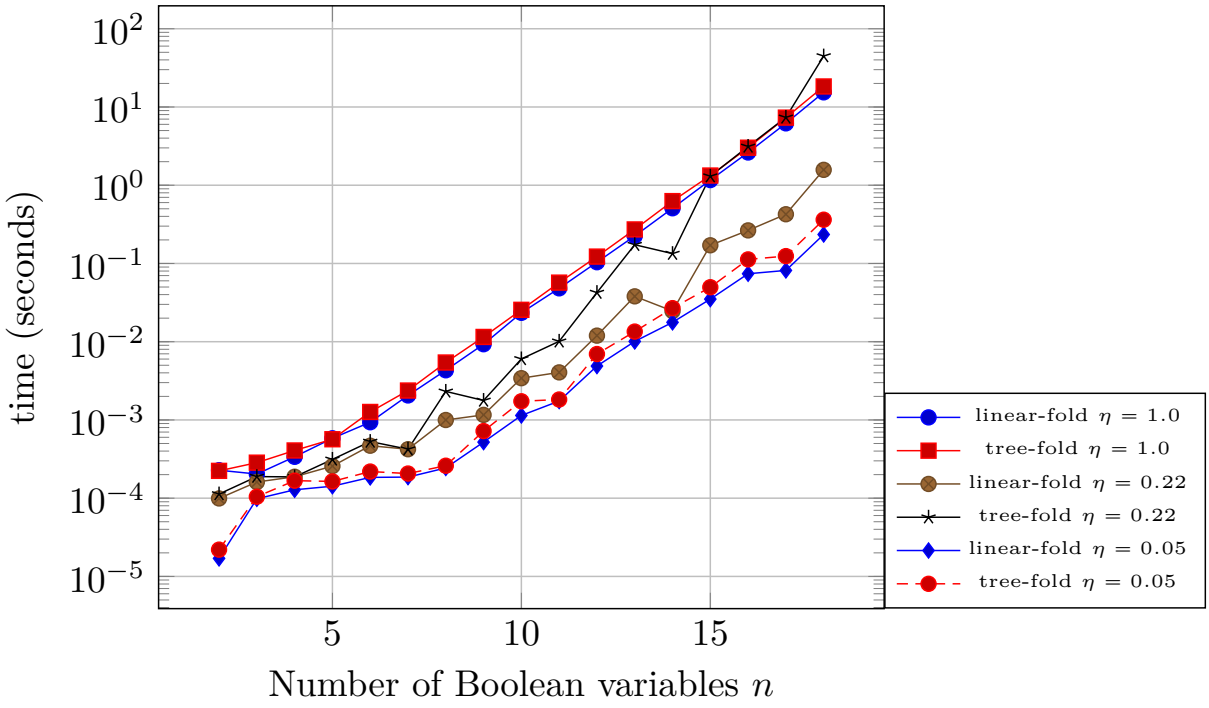


Figure 5.11: Comparison of ROBDD construction time using linear and tree-like fold operations. The times are plotted for various values of η.

Implementation 5.24 (Var-args versions of BDD operations, `and`, `or`, `xor`, `and-not`).

```
(defun bdd-xor-list (&rest bdd-list)
  (reduce #'bdd-xor bdd-list :initial-value *bdd-false*))

(defun bdd-or-list  (&rest bdd-list)
  (reduce #'bdd-or bdd-list :initial-value *bdd-false*))

(defun bdd-and-list (&rest bdd-list)
  (reduce #'bdd-and bdd-list :initial-value *bdd-true*))

(defun bdd-and-not-list (bdd1 &rest bdd-list)
  (bdd-and-not bdd1
               (apply #'bdd-and-list bdd-list)))
```

5.6 Generating randomly selected ROBDD of n variables

Minterm index	x_4	x_3	x_2	x_1	F	Minterm
0	0	0	0	0	1	$(\neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \neg x_4)$
1	0	0	0	1	0	
2	0	0	1	0	0	
3	0	0	1	1	1	$(x_1 \wedge x_2 \wedge \neg x_3 \wedge \neg x_4)$
4	0	1	0	0	0	
5	0	1	0	1	0	
6	0	1	1	0	0	
7	0	1	1	1	0	
8	1	0	0	0	0	
9	1	0	0	1	1	$(x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4)$
10	1	0	1	0	0	
11	1	0	1	1	0	
12	1	1	0	0	0	
13	1	1	0	1	0	
14	1	1	1	0	0	
15	1	1	1	1	0	

Figure 5.12: 4-Variable truth table representing $521_{10} = 0000001000001001_2$. To interpret a binary integer as a truth table, enter the bits in order with least significant bit at the top and most significant bit at the bottom. Bits which are 1 correspond to minterms 0, 3, and 9.

In Section 6.1.3 we discuss an experiment we made of measuring properties of randomly selected ROBDDs. In this section (5.6) we discuss the algorithm we use for generating a randomly selected ROBDD of n variables.

Any Boolean function of n variables is uniquely determined by a truth table having 2^n rows. To randomly select such a Boolean function, we need only choose a random number having 2^n bits; *i.e.* a number between 0 and $2^{2^n} - 1$. As shown in Example 5.25, the 1's in that number determine which minterms are present in the DNF.

Example 5.25 (Randomly selected DNF, determining a Boolean function of n variables.)**.**
For example, $0 \leqslant 521 < 2^{2^4} = 65536$. So the integer

$$521_{10} = 1 \cdot 2^0 + 1 \cdot 2^3 + 1 \cdot 2^9 = 0000001000001001_2$$

represents the truth table shown in Figure 5.12. From this truth table we generated the Boolean expression

$$((\neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \neg x_4 \wedge \neg x_5) \vee (x_1 \wedge x_2 \wedge \neg x_3 \wedge \neg x_4 \wedge \neg x_5) \vee (x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4)),$$

having minterms 0, 3, and 9.

From the DNF of the Boolean expression, we generated the corresponding ROBDD as explained in Section 5.5.

An optimization which we make is that when we need only the ROBDD, and neither the explicit truth table nor the full DNF, we may generate the integer lazily; *i.e.* we may generate a lazy stream of 2^n bits, and maintain a count of the index of the minterm. When 1 is read from the stream, generate the minterm, and use a fold operation [Hut99] as explained in Section 5.5. As long as the random number generator is sufficient, the optimization is valid.

We are assuming that generating 2^n bits with 2^n successive calls to `(random 2)` generates effectively the same distribution as extracting the bits from an integer generated from a single call to `(random (expt 2 (expt 2 n)))`. SBCL (version 1.4.3) uses the MT19937 `prng` (pseudo random number generator) algorithm [MN98] for generating pseudo random numbers. The period of the random number generator is $2^{19969} - 1$ [Bou18], which means we would need to call `random` 1000 times per second for much longer than the age of the universe (1.8×10^{303} seconds.) to exhaust its period.

In Section 6.1.3 we discuss the algorithm of generating multiple ROBDDs and counting their nodes in order to obtain an estimate for the size distribution of the set of all n-variable ROBDDS. As mentioned above, this process is parallelizable. In performing such an experiment we took care to detect whether two compute nodes

(two machines in the compute cluster) select the same integer. Such an occurrence could skew the distribution measurements.

If integers are chosen at random in the space from $0 \leqslant i < 2^{2^n}$, the chances of choosing the same number twice are minuscule to the point of being negligible, even on separate processes on separate machines, assuming the program is bug-free. However, we had a bug in an early version of our program where we neglected to seed the random number generator appropriately. Because of this bug, we generated the exact same sequence of integers on every node in the cluster. Because of this experience, we thereafter became arguably overly cautious.

In order to assure we never count the same ROBDD twice, each process creates a log file with three columns: n, measured node count of the ROBDD, and the base 36 encoded integer representing the truth table. The reason we chose base 36 is because the Common Lisp function, `format`, can easily generate such a representation, and the Common Lisp `read` function can easily parse it. Example 5.26 shows Common Lisp code for reading and writing base 36.

Example 5.26 (Reading and writing base 36 integers)**.**

```
(with-output-to-string (stream)
  (format stream "~36R ;" 415558567100094532))
;; returns the string "35NR7U935D1G ;"

(let ((*read-base* 36))
  (with-input-from-string (stream "35NR7U935D1G")
    (read stream)))
;; returns the integer 415558567100094532
```

We terminate each line with a semicolon in order to detect prematurely terminated lines. A line might be incomplete in the case that the compute cluster manager[1] has automatically killed our process after a time-out is reached. In the case that the Common Lisp process is terminated during a call to `format`, we may encounter such a log file where the base 36 integer was not completely written. Granted, this is unlikely in the case of $n = 8$, but the length of the printed representation of an integer $i > 0$ (in any base b, including base 36) is bounded by an upper limit proportional to $\log_b i$. If $0 < i < 2^{2^n}$, then

$$\log_{36} i < \log_{36} 2^{2^n} = 2^n \cdot \log_{36} 2\,.$$

This means that a base 36 representation of the truth table of an 18-variable Boolean formula may have $2^{19} \cdot \log_{36} 2 \approx 101,411$ characters. It is indeed conceivable that the cluster manager might kill the process while it is busy writing a 100 kilobyte line.

Another advantage of terminating each line with a semicolon is that a program which only wishes to read the first two values of each line (*i.e.*, n and the corresponding ROBDD size) may skip the rest of the line simply by reading characters until a semicolon is reached, thus obviating the need to allocate a large bignum.

Example 5.27 (File logging the discovered sizes of several 8-variable ROBDDs)**.**

```
8 76 3HYKNOKIES210SIUF8EDK73P345TMJIEHAMM4SU1377EE6MZ2C ;
8 77 3FPE2XYQEDV5P8RUJIVANLRODX920RAD8MKFJMEF6SOS6TUBOR ;
8 75 1NIYCE4MQBCFI6FO7F3WIFLUOBOPN1WF5SS12BZBZ4VI72NWN6 ;
8 77 D6S4O7RY5ST4SNIUV8CIGEOI2TTGVXGSGIO4RQITCM7VN91QJ ;
8 76 6868C2K8WMXDT5WM1YR4DG698VRSOZC9HFCU6J4U8RYOIKYK8D ;
8 72 644RONLW7O9FPNRP9U6ONC4N8X7BK7DQ2MOD88ONECAN36P7UB ;
8 74 TH4WLQZPMWYWUDRNYBAPQARM5BP9IP5PXHSN9BEME8KS5Q8P ;
8 75 2LHRTNM41J7MFQZI6A4O9GQJJUZJ3U3ZPETONZEV3EEILH7B3N ;
8 75 4QJG869771XXQ4PTMK1MQ5J58RWF5HWO35QUGZ6G9P76LKEV2I ;
8 72 4WCY1O43GHSP153B2RV5HBS5ZPBTAQWC6ZXIXG5X6NBSS8FJD ;
```

[1]The cluster manager is a program which runs on the cluster with administrator rights. It manages scheduling and starting jobs requested by all the users. Each job submitted is accompanied by an estimate of the maximum time required. If the job runs for more than that time-limit, the cluster manager eventually abruptly kills the job.

We might have also considered base 64 encoding which would have reduced the file size by an additional 14%. While the integer $2^{2^{19}}$ requires 101+ kilobytes to print in base 36, it only requires 87+ kilobytes to print in base 64. If $x > 0$,

$$\frac{log_{64}x}{log_{36}x} \approx 0.863 \approx 1 - 14\% \,.$$

We chose base 36, because it is the maximum base supported by the Common Lisp `format` function. We could have used an external library, or implemented our own base 64 encoding, but chose not to do so.

After multiple parallel processes have finished and produced their log files as in Example 5.27 it remains to remove duplicates. Each individual file can be uniqified using the UNIX `sort` program:

```
sh> mv file-1 tmp
sh> sort -u -T $PWD tmp > file-1
```

And once all the individual files have been sorted and are free of duplicates, they can be combined with another call to `sort` using the `-m` option. The `-m` option instructs `sort` to simply merge the input files which are assumed to already be sorted. Such a merge is an $O(n)$ operation whereas sort is normally $\Omega(n \log(n))$. (We refer the reader to Wegener [Weg87, Section 1.5] for a discussion of Ω notation.)

```
sh> sort -m -u -T file-1 file-2 file-3 ... > file-combined
```

5.7 Conclusions and perspectives

In Section 5.4 we explained two flavors of the fold operation for use in multiple-argument ROBDD Boolean operations. We mentioned that the results were surprising.

In situations where the size of the ROBDD grows during the fold operation, intuition would lead us to believe that performing more operations on small ROBDDs is faster than few operations on large ROBDDs, especially in extreme situations where sizes are growing exponentially. In such situations, once intermediate operations are finished, their memory resources could be returned to the system via garbage collection. We believe more investigation is needed to characterize the situations in which each strategy is more effective.

Chapter 6

Numerical Analysis of the Worst-Case Size of ROBDDs

In Chapter 5 we introduced a special flavor of Binary Decision Diagram called the ROBDD, examining its construction, some of its properties, and discussed its usefulness in Boolean algebra. We intend to extend the theory in Chapter 7 to provide a data structure for representing Common Lisp types, and thereafter proceed by using it to improve the solutions to the problems introduced in Chapter 4. Before looking at the relation of ROBDDs to Common Lisp types, we first wish, in the current chapter, to examine some of the size-related properties of the ROBDD.

Even though the ROBDD is a lightweight data structure to implement (*i.e.*, it can be easily implemented programmatically), some of its behavior regarding the amount of necessary memory allocation may not be obvious in practice. In this report we convey an intuition of expected sizes and shapes of ROBDDs from several perspectives.

Section 6.1.2 examines worst-case sizes of ROBDDs: first, we look exhaustively at cases involving a small number of variables; then, we examine experimentally the average and worst-cases sizes for several cases involving more variables. Section 6.1 examines the shapes of the graphs of the worst-cases sizes. In Section 6.1.7, we use an intuitive understanding to derive an explicit formula to calculate the worst-case size for a given number of variables. Finally, in Section 6.2, we provide an algorithm for generating a worst-case sized ROBDD for a given number of Boolean variables.

6.1 Worst-case ROBDD size and shape

The BDD shown in Figure 5.2 is a 31 nodes UOBDD, reduced in Figure 5.5 to an 8 nodes ROBDD, thanks to the three reduction rules presented in Section 5.1.2. We may naturally ask whether this reduction process is typical.

The size and shape of a reduced BDD depends on the chosen variables ordering [Bry86]. Finding the best ordering is coNP-Complete [Bry86]. In this report, we do not address the questions of choosing or improving the variable ordering. Given a particular variables ordering however, the size and shape of the ROBDD depends only on the truth table of the Boolean expression. In particular, it does not depend on the chosen representation for the expression. For example, $(A \vee B) \wedge C$ has the same truth table as $(A \wedge C) \vee (B \wedge C)$, so these two expressions are equivalent and will be reduced to the exact same ROBDD. In a practical sense, the ROBDD serves as a canonical form for Boolean expressions.

The best-case size (in terms of node count) for a constant expression is obviously one, *i.e.*, a Boolean expression which is identically $\top$ or $\bot$. But what is the worst-case size of an ROBDD of n variables? We examine this question both experimentally and theoretically in the following sections.

6.1.1 Process summary

We start by showing all possible ROBDDs for the 1- and 2-variable cases. Then, we address the question of the worst-case size by looking at the exhaustive list of ROBDDs up to the 4-variable case, extrapolating from random samples thereafter. The way we do random sampling is also explained.

Given the above data, we observe that the difference between the worst-case size and the average size becomes negligible as the number of Boolean variables increases. At this stage however, this observation is only a conjecture, and we would like to prove it formally. We define a quantity called residual compression ratio which measures the effectiveness of the ROBDD representation, as compared to the size of the truth table. We note from experimental data that this ratio decreases, but to which value is unclear.

We continue by deriving a formula for the worst-case ROBDD size, based on the number of nodes in each row, and holding for any number of variables. This derivation is motivated by images of sample worst-case ROBDDs as the number of variables increases. The derivation is obtained as follows.

First, we introduce a threshold function which represents the competition between an increasing exponential and a decreasing double exponential. We are able to express the worst-case ROBDD size in terms of this threshold. Then, we argue that natural number thresholds are non-decreasing, and that real number thresholds are strictly increasing. We then derive bounds on the threshold function, and use them to provide an algorithm for computing threshold values, usually within one or two iterations.

Ultimately, we use those bounds to show that the residual compression ratio indeed tends to zero.

6.1.2 Experimental analysis of worst-case ROBDD Size

We saw above that, given a variable ordering, every truth table of n variables corresponds to exactly one ROBDD. Otherwise stated, there is a one–to–one correspondence from the set of n-variable truth tables to the set of n-variable ROBDDs. However, for any given truth table, there are infinitely many equivalent Boolean expressions. An n-variable truth table has 2^n rows, and each row may contain a $\top$ or $\bot$ as the expression's value. Thus, there are 2^{2^n} different n-variable truth tables.

For small values of n it is reasonable to consider every possible ROBDD exhaustively, to determine the maximum possible size. However, it becomes impractical to do so for large values. For example, there are $2^{2^{10}} > 1.80 \times 10^{308}$ ROBDDs of 10 variables. In our analysis, we treat the 1- through 4-variable cases exhaustively, and use extrapolated results (explained below) otherwise.

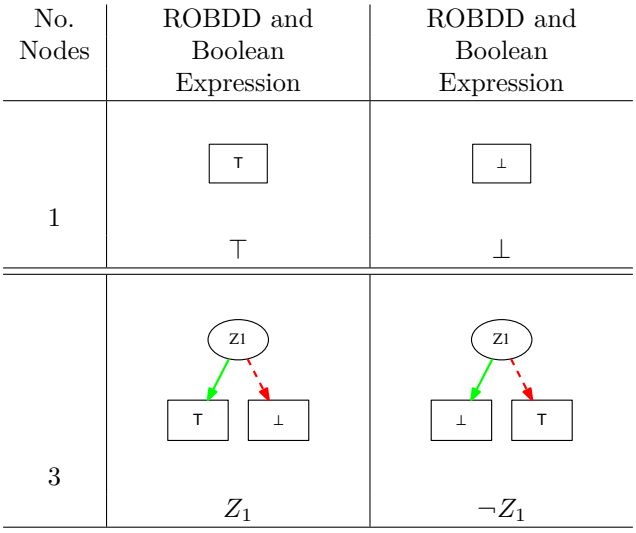


Figure 6.1: All ROBDDs of one variable

Figure 6.1 shows all the possible ROBDDs of a single variable. We see that only 4 ROBDDs are possible ($2^{2^n} = 2^{2^1} = 4$). Two of the ROBDDs have one node, and the other two have two nodes. By convention, we consider an n-variable expression as an expression having n or fewer variables. We use this convention because some Boolean expressions of n-variables can be reduced to equivalent expressions having fewer variables. For example, $A \vee (A \wedge \neg B)$, a 2-variable expression, is in fact equivalent to just A. Figure 6.2 shows an exhaustive list of the possible ROBDDs of 2 variables. Here, the worst-case node count is 5, occurring twice out of a total of $2^{2^n} = 2^{2^2} = 16$ possible expressions.

6.1.3 Statistics of ROBDD size distribution

Definition 6.1. Let us define $\mathscr{H}_n(x)$ as the number of Boolean functions of n-variables whose ROBDD contains exactly x nodes. If there are no n-variable ROBDD with exactly x nodes (including when x is not an integer), then we define $\mathscr{H}_n(x)$ by straightforward interpolation. *I.e.*, let x_0 be the maximum integer, $x_0 < x$, such that there exists an n-variable Boolean equation whose ROBDD has exactly x_0 nodes; and let x_1 be the minimum integer, $x_1 > x$, such that there exists an n-variable Boolean equation whose ROBDD has exactly x_1 nodes; then

$$\mathscr{H}_n(x) = \mathscr{H}_n(x_0) + \frac{\mathscr{H}_n(x_1) - \mathscr{H}_n(x_0)}{x_1 - x_0} \cdot (x - x_0)\,.$$

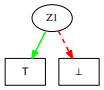
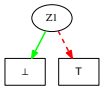
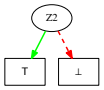
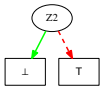
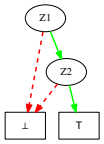
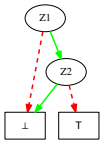
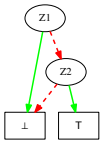
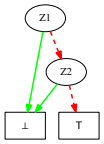
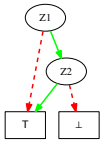
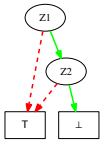
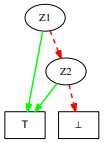
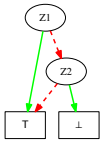
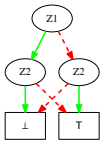
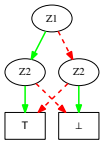
No. Nodes	ROBDD and Boolean Expression	ROBDD and Boolean Expression	ROBDD and Boolean Expression	ROBDD and Boolean Expression
1	 $\top$	 $\bot$		
3	 Z_1	 $\neg Z_1$	 Z_2	 $\neg Z_2$
4	 $(Z_1 \wedge Z_2)$	 $(Z_1 \wedge \neg Z_2)$	 $(\neg Z_1 \wedge Z_2)$	 $(\neg Z_1 \wedge \neg Z_2)$
4	 $((Z_1 \wedge Z_2) \vee \neg Z_1)$	 $((Z_1 \wedge \neg Z_2) \vee \neg Z_1)$	 $((\neg Z_1 \wedge Z_2) \vee Z_1)$	 $((\neg Z_1 \wedge \neg Z_2) \vee Z_1)$
5	 $((Z_1 \wedge \neg Z_2) \vee (\neg Z_1 \wedge Z_2))$	 $((Z_1 \wedge Z_2) \vee (\neg Z_1 \wedge \neg Z_2))$		

Figure 6.2: All ROBDDs of two variables

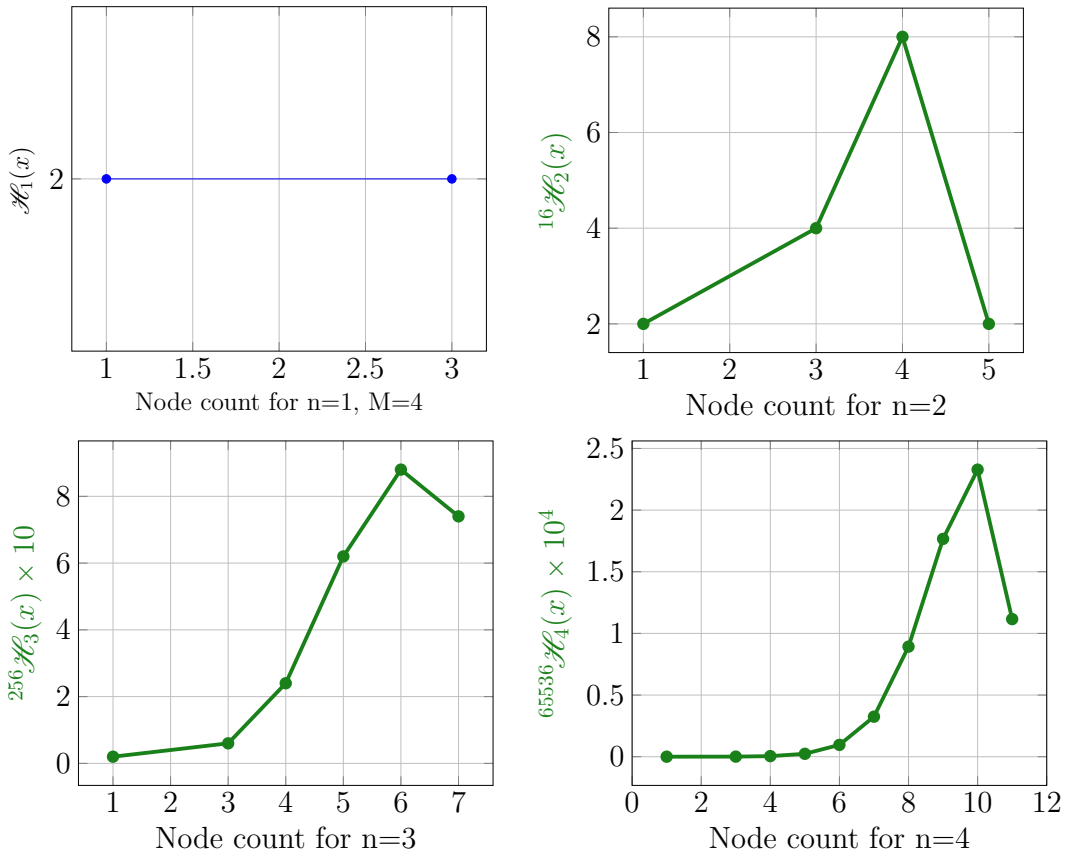


Figure 6.3: Histograms illustrating size distributions of ROBDDs from 1 to 4 Boolean variables. The histograms are based on exhaustive data.

Definition 6.2. $\overline{\mathscr{H}}$ denotes the histogram function normalized by the total number of Boolean functions which exist for the given number of variables.

$$\overline{\mathscr{H}_n}(x) = \frac{\mathscr{H}_n(x)}{2^{2^n}}$$

Note that $\mathscr{H}_n$ differs from the actual histogram in two subtle ways: (1) A histogram is a discrete function of integers to integers, whereas $\mathscr{H}_n$ is a continuous function of reals to reals; and (2) in the case that there are no ROBDDs of size x, the value of the histogram would be 0, but the value of $\mathscr{H}_n$ is interpolated from the closest nearby values by the formula indicated in Definition 6.1.

Figure 6.3 is a plot of $\mathscr{H}_1$ through $\mathscr{H}_4$. In Figure 6.3, we exhaustively counted the possible sizes of each ROBDD for 1 to 4 variables. Since $\mathscr{H}_4(11)$ is the right-most point on the plot, the worst case for 4 variables is 11 nodes. We can estimate from Figure 6.3 that of the 65536 different Boolean functions of 4 variables, only about 12000 of them (18%) have node size 11, $\mathscr{H}_{11} \approx 1200$. The average size is about 10 nodes.

When $n > 4$, we will estimate the plots of $\mathscr{H}$ by extrapolation. We take a large number, M, of samples of randomly chosen Boolean functions. In such case, we use the following notation:

Notation 6.3. We denote $^M\mathscr{H}_n(x)$ to indicate that the histogram, $\mathscr{H}_n(x)$, has been estimated by M unique samples. In the case that $M = 2^{2^n}$ we have $^M\mathscr{H}_n = \mathscr{H}_n$.

In Figure 6.4, we have extrapolated from random sampling the 5 through 10-variable cases as described below.

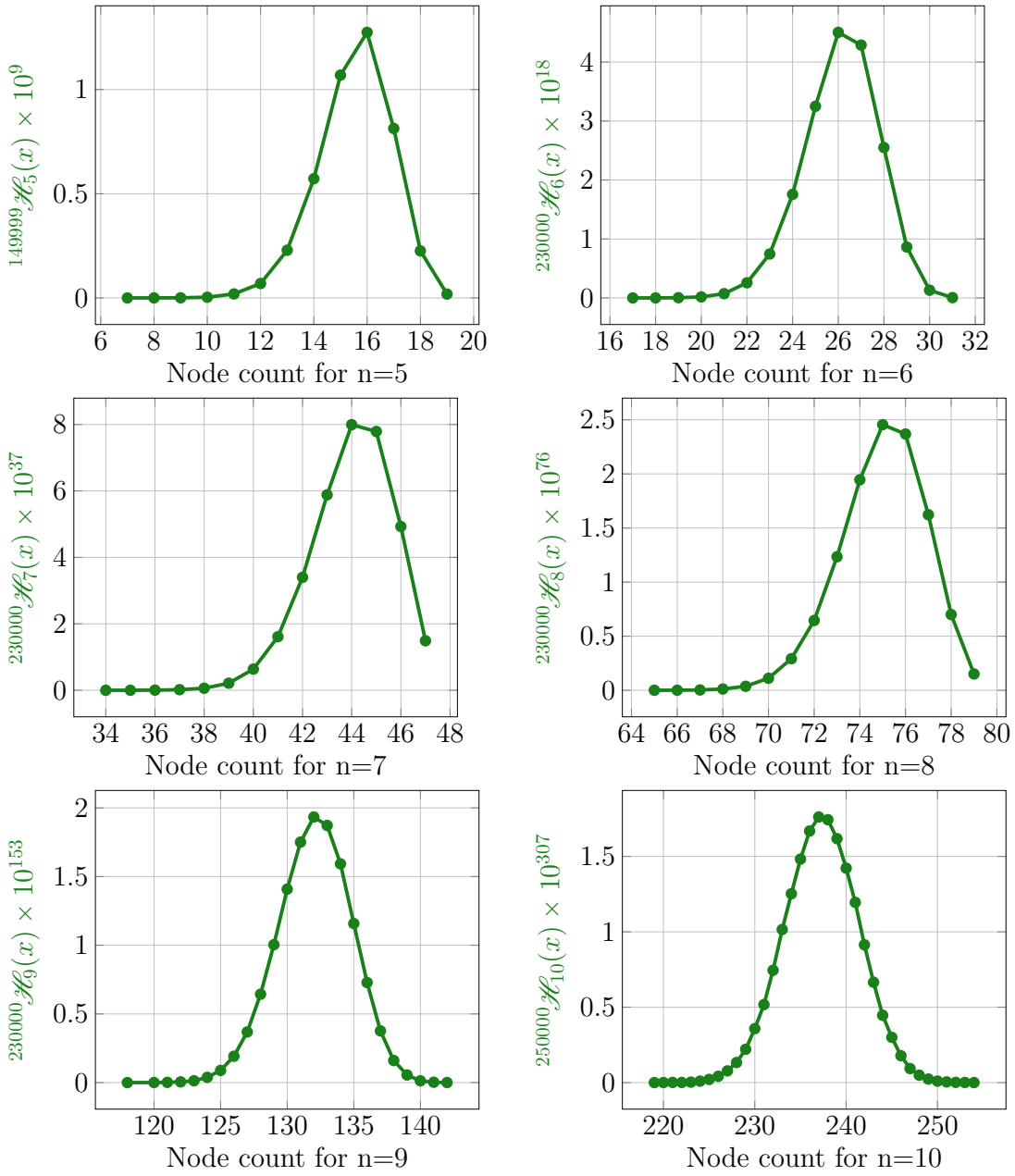


Figure 6.4: Histograms illustrating size distributions of ROBDDs from 5 to 10. The histograms are based on extrapolations from sampled data. The plots show $y = {}^{M}\mathscr{H}_n$ vs $x =$ ROBDD size, for $5 \leqslant n \leqslant 10$, and in each case with the maximum value of M from our experiments.

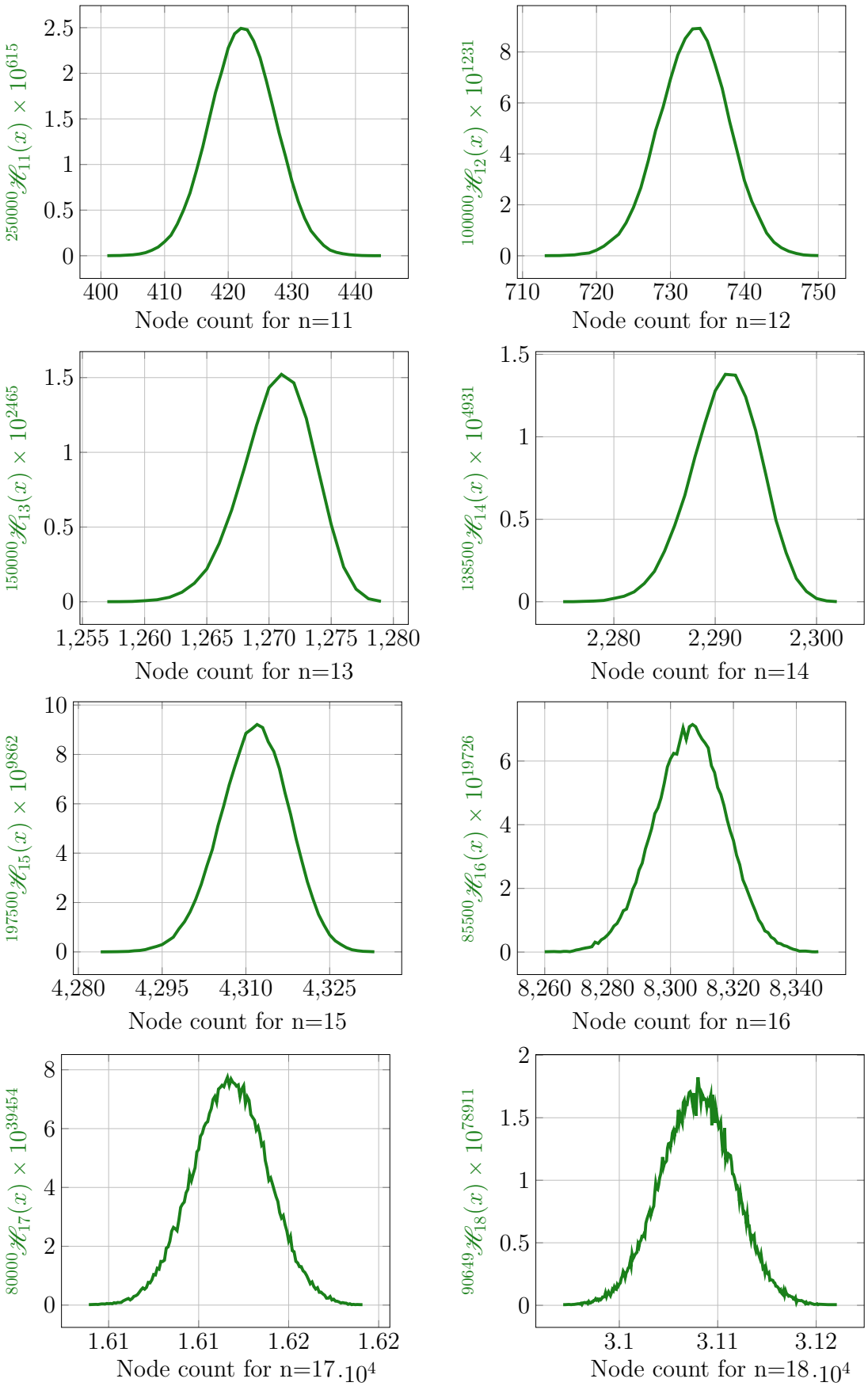


Figure 6.5: Histograms illustrating size distributions of ROBDDs from 11 to 18. The histograms are based on extrapolations from sampled data. The plots show $y = {}^{M}\mathscr{H}_n$ vs x = ROBDD size, for $11 \leqslant n \leqslant 18$, and in each case with the maximum value of M from our experiments.

No. Variables (n)	No. Samples (M)	No. Unique Sizes
5	149,999	13
6	230,000	15
7	230,000	14
8	230,000	15
9	230,000	24
10	250,000	36
11	250,000	46
12	100,000	38
13	150,000	24
14	138,500	29
15	197,500	53
16	85,500	93
17	80,000	163
18	90,649	289

Figure 6.6: Number of samples, M, used for generating the plots in Figures 6.4 and 6.5. The table also shows the number of unique ROBDD sizes which were detected for each value of n.

We generated the data in Figure 6.4 for 5 through 8 Boolean variables, by randomly selecting truth tables in the manner described in Section 5.6, counting the nodes in the ROBDD, and multiplying by a factor to compensate for the sample size. We did the computation work in Common Lisp [Ans94] using the SBCL [New15] Common Lisp compiler.

The number of samples we take while constructing the plots in Figure 6.4, is constrained by the computation-time at our disposal. Construction of such large ROBDDs is compute intensive but parallelizable. We have shared access to a cluster of Intel Xeon^{TM} E5-2620 2.00GHz 256GB DDR3 machines. Consequently, we tried to achieve a reasonably large sample size with the resources available.

Figure 6.6 lists the number of samples per value of n. See Section 6.1.4 for the discussion of how we determined which number of samples to use. Figure 6.7 consolidates the data from Figures 6.3 and 6.4 into a single plot but normalized, so that the total number of Boolean functions in each curve is 100%. This normalization allows us to interpret a point (x, y) on the curve corresponding to n variables as meaning that a randomly selected Boolean expression of n-variables has probability y of having an ROBDD which contains exactly x nodes.

Each point, (n, σ_n), in the plot in Figure 6.9 was calculated from a corresponding curve $\mathscr{C}_n$ of Figure 6.7 by the formula in Notation 6.4.

Notation 6.4. The standard deviation of a distribution defined by a histogram is calculated by the formula:

$$\sigma_n = \sqrt{\sum_{(x,y)\in\mathscr{C}_n} y \cdot (x - \mu_n)^2}\,, \quad \text{with} \quad \mu_n = \sum_{(x,y)\in\mathscr{C}_n} x \cdot y\,.$$

It is not clear from Figure 6.7, whether the spread of ROBDD size grows with the number of variables. However, from the standard deviation plot in Figure 6.9, the spread seems to grow in absolute terms. Despite this apparent spread, the average (expected size), median, and worst-case sizes summarized in Figure 6.8 give the impression that the distinction between average size and worst-case size becomes negligible as the number of variables increases. Otherwise stated, it appears that for large values of n, $|ROBDD_n|$ becomes a good approximation for average size, an observation which seems related to the Shannon effect as discussed by Gröpl *et al.* [GPS98].

The plot in Figure 6.7 gives the impression that the distribution of possibles ROBDD sizes for a given number of variables, is clustered around the average such value. The standard deviation plot in the same figure gives an impression of how tight this clustering is. In this report, we don't present a formula for this standard deviation as a function of n, but from observing the plot, one would suspect that it grows faster than linearly.

One might be tempted to assume that the data represented in Figure 6.4, and consequently in Figure 6.7, follows a normal distribution, as the curves have a bell-like shape. However, the distribution is not Gaussian. In particular, each of the curves extend left to the point $(1, 2)$ because there are always two constant functions

of N variables, namely, $f = \top$ and $f = \bot$. On the other hand, we did not see any case in our experimentation where the curves extended to the right any considerable distance beyond the peak. Later, we show what the actual maximum size of an ROBDD of N variables is (see Figure 6.33), and in each case, the rightmost points in Figure 6.7 agree impeccably with Figure 6.33.

If we believed the data followed a Gaussian distribution, we could interpret the standard deviation more strictly. But for any distribution where we can calculate the mean and standard deviation, we can interpret the standard deviation according to the Chebyshev inequality. The standard deviation plot (Figure 6.9) can be interpreted according to the Chebyshev inequality [Als11], with X being the size of a randomly selected ROBDD.

$$\Pr(|X - \mu| > k \cdot \sigma) \leqslant \frac{1}{k^2} \qquad \text{Chebyshev's inequality}$$

If the standard deviation of the probability function (Figure 6.7) for n Boolean variables is σ_n and the average ROBDD size is μ_n, then for a given real number, $k \geqslant 1$ ($k > 1$ in practice), the probability of a randomly selected ROBDD of n-variables having more than $\mu_n + k \cdot \sigma_n$ nodes or less than $\mu_n - k \cdot \sigma_n$ nodes, is less than $\frac{1}{k^2}$. As an example of using the plots in Figures 6.8 with the Chebyshev inequality, taking $k = 2$:

$$\begin{aligned}
\mu_8 &= 75.0 && \text{from Figure 6.8}\\
\sigma_8 &= 1.89 && \text{from Figure 6.9}\\
k &= 2\\
\frac{1}{k^2} &= \frac{1}{2^2} = 25\%\\
\mu_8 - k \cdot \sigma_8 &= 71.22\\
\mu_8 + k \cdot \sigma_8 &= 78.78\,.
\end{aligned}$$

This means that given a randomly selected 8-variable ROBDD, there is a $100\% - 25\% = 75\%$ chance that it has between 71 and 79 nodes.

6.1.4 Sufficiency of sample size

In Section 6.1.3, we discussed the approximation of ROBDD size histograms by a method of extrapolation based on a random sample. The size of any such sample is bound to be miniscule compared to the size of the space of Boolean functions. For example, there are $2^{2^{10}} = 1.8 \times 10^{308}$ Boolean functions of 10 variables. In order to sample just 1% of this space, even at a rate of 1000 per second, we need 1.8×10^{303} seconds. Assuming the age of universe is 13.8 billion years that comes out to only 4.4×10^{17} seconds. Knowing that we cannot sample a significant number of Boolean functions, how can we know that whether we have sampled a sufficient amount, to have confidence that the histograms in Figures 6.4 and 6.5 are good approximations?

To answer this question, we once again take a look at the Chebyshev inequality. Our confidence in its results is limited by our confidence that the approximated average, μ_n, and standard deviation, σ_n, are close to the actual result. *I.e.*, if we increased the sample size, μ_n and σ_n would change significantly?

For illustration, we will precisely examine the plots in Figures 6.4 and 6.5. Figures 6.10, through 6.15 show some of the corresponding plots, but illustrated considering fewer samples. We observe, not surprisingly, that as the number of samples increases (approximately doubles with each successive plot), the distribution curve tends to become smoother. That the distribution curve tends to smoothen as M (the sample size) increases suggests that the sequence of curves is converging to some limiting curve as M increases. We do not explicitly assume this but the general shape of the curves suggests that this limit curve is a normal distribution function.

Notation 6.5.

$$\phi(\mu, \sigma^2) = \frac{M}{\sqrt{2\pi\sigma^2}} e^{-\frac{x-\mu}{2\sigma^2}}\,.$$

To investigate how closely the sampled curves approximate a normal distribution, we must extract the mean and standard deviation (μ and σ). Of course the calculated values of μ and σ will certainly depend on the actual samples which have been chosen randomly.

Figure 6.16 shows the mean values (μ_n with n selected from 5 to 18) and the corresponding standard deviations (σ_n), both as function of M, the number of samples taken. We observe that the mean values (*i.e.*

the average ROBDD size for n Boolean variables) and the standard deviations do not seem to converge to any asymptotic values, but neither μ_n nor σ_n vary significantly. Figure 6.16 shows that for the samples we made, the total excursion of μ is less than 1% and the total excursion of σ is less than 4%.

For a visual illustration of how closely each of the plots in Figures 6.10 through 6.15 agree with the theoretical normal distribution, we present Figures 6.18 through 6.22 which superimpose the sampled curves over the normal distribution whose μ and σ values are also shown in the respective figures. Also, as a summary, Figure 6.16 shows averages and standard deviations associated with each population size M.

While the method of superimposing the sampled curves over the corresponding Gaussian distributions gives a visual illustration with which to measure the sufficiency of the sample size, it does not seem to help numerically. Figures 6.23 through 6.27 illustrate the histogram difference functions.

Definition 6.6. The following denotes the arithmetic difference between two histograms, ${}^{M}\mathscr{H}_n$, of successive sample sizes, M and $2M$.

$$\Delta^{2M}\mathscr{H}_n(x) = {}^{2M}\mathscr{H}_n(x) - {}^{M}\mathscr{H}_n(x)$$

The figures also show, the L_2 norm of the difference function calculated between successive samples.

Notation 6.7.

$$\begin{aligned}\|\Delta^{M}\mathscr{H}_n\|_2 &= \sqrt{\int_1^{|ROBDD_n|} \left({}^{M}\mathscr{H}_n(x) - {}^{\frac{M}{2}}\mathscr{H}_n(x)\right)^2 \, dx} \\ &= \sqrt{\int_1^{|ROBDD_n|} \left(\Delta^{M}\mathscr{H}_n(x)\right)^2 \, dx}\end{aligned}$$

We observe (Figures 6.23 through 6.27) that in most cases the value of the integral trends downward towards zero. This supports our supposition that the sequence of functions is Cauchy, and is thus convergent.

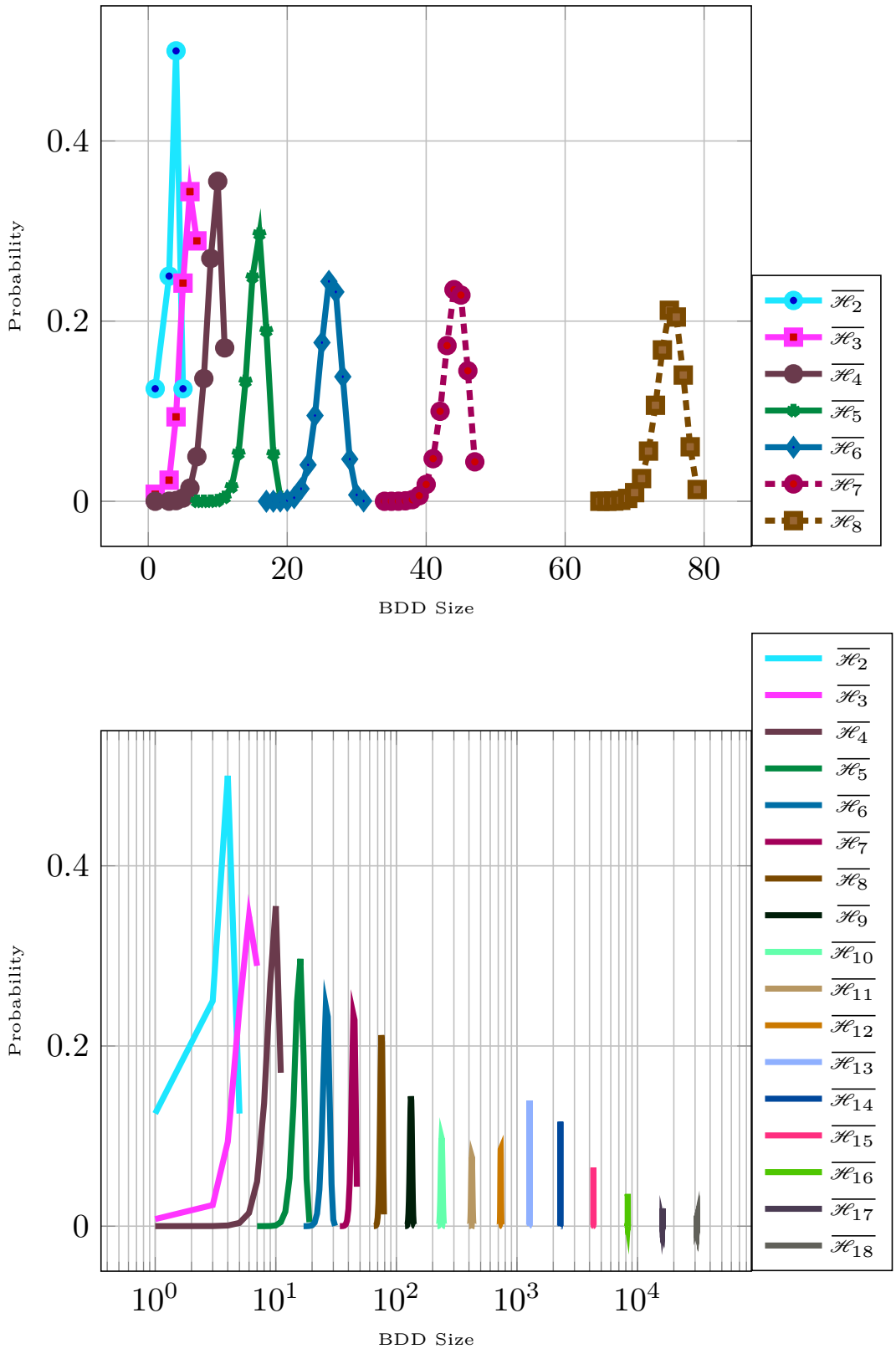


Figure 6.7: Normalized histograms of size distribution probability functions for ROBDDs of 2 to 10 variable Boolean expressions, based on exhaustive data for 2, 3 and 4 variables, and on randomly sampled data for 5 and more variables.

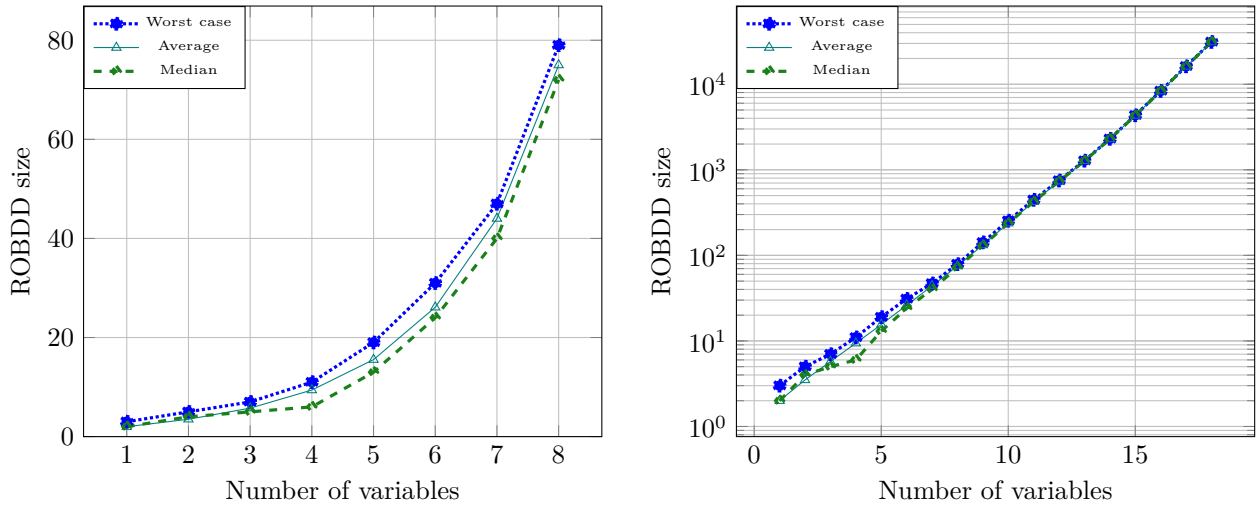


Figure 6.8: Expected and worst-case ROBDD size from 1 to 10 variables, exhaustively determined for 1 through 4 variables, experimentally determined for 5 and more variables.

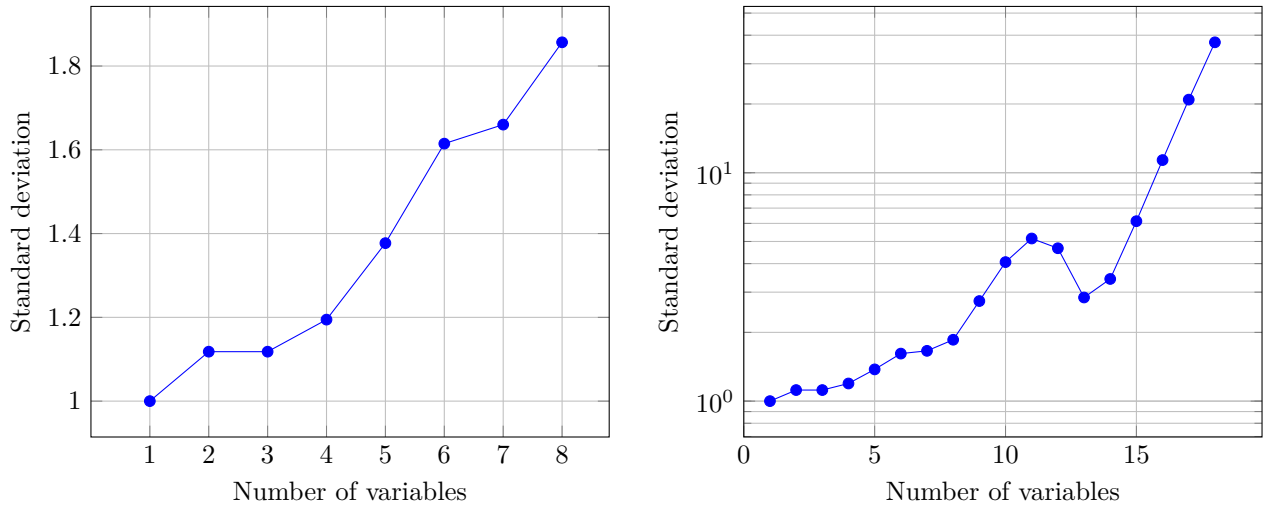


Figure 6.9: Standard deviations for each of the curves shown in Figure 6.7 and whose averages are shown in Figure 6.8.

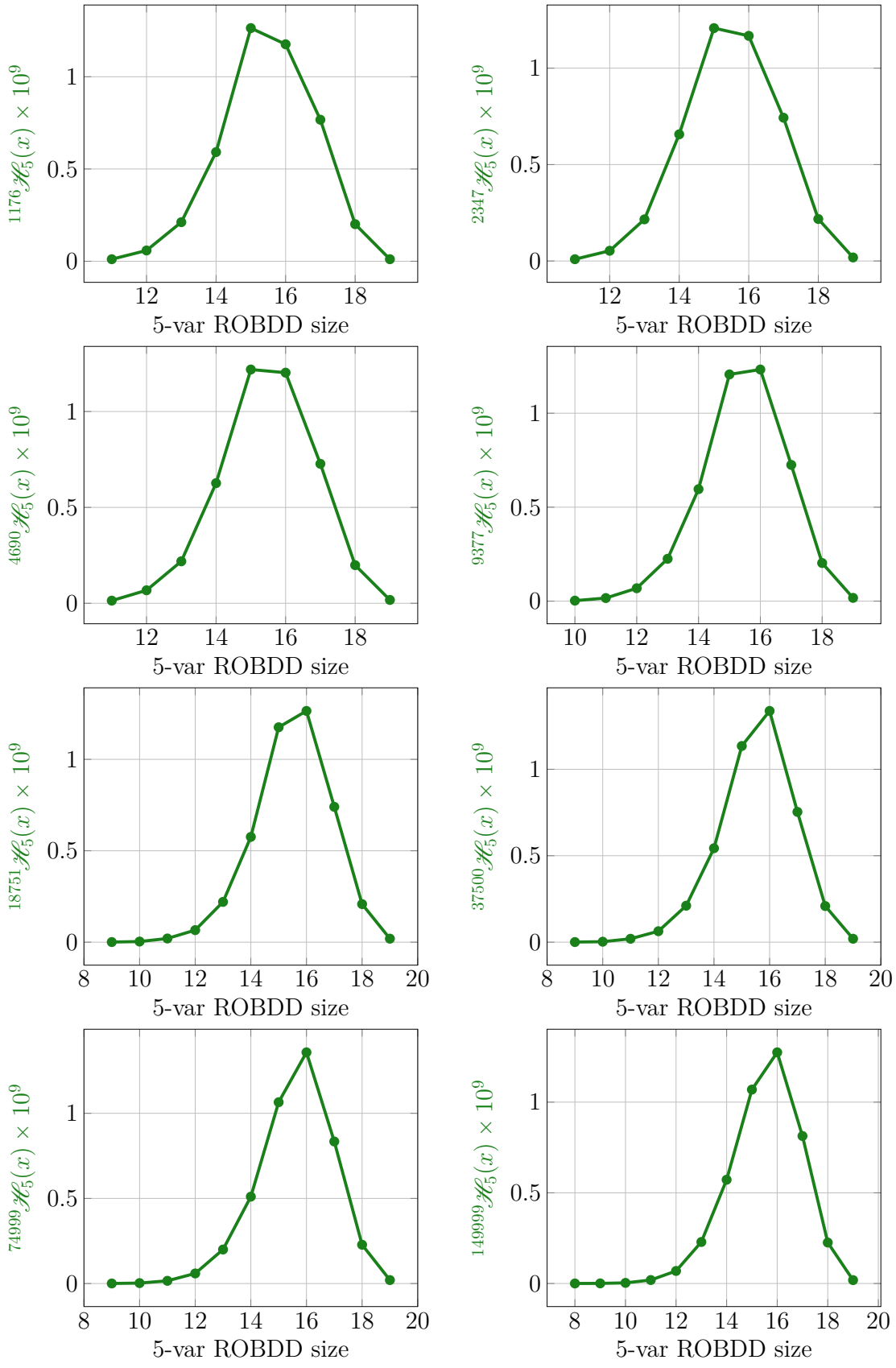


Figure 6.10: Histograms illustrating size distributions of successively larger samples for 5 variables. Each plot shows $y = {}^{M}\mathscr{H}_5$ for successively larger values of M *vs.* x = ROBDD size.

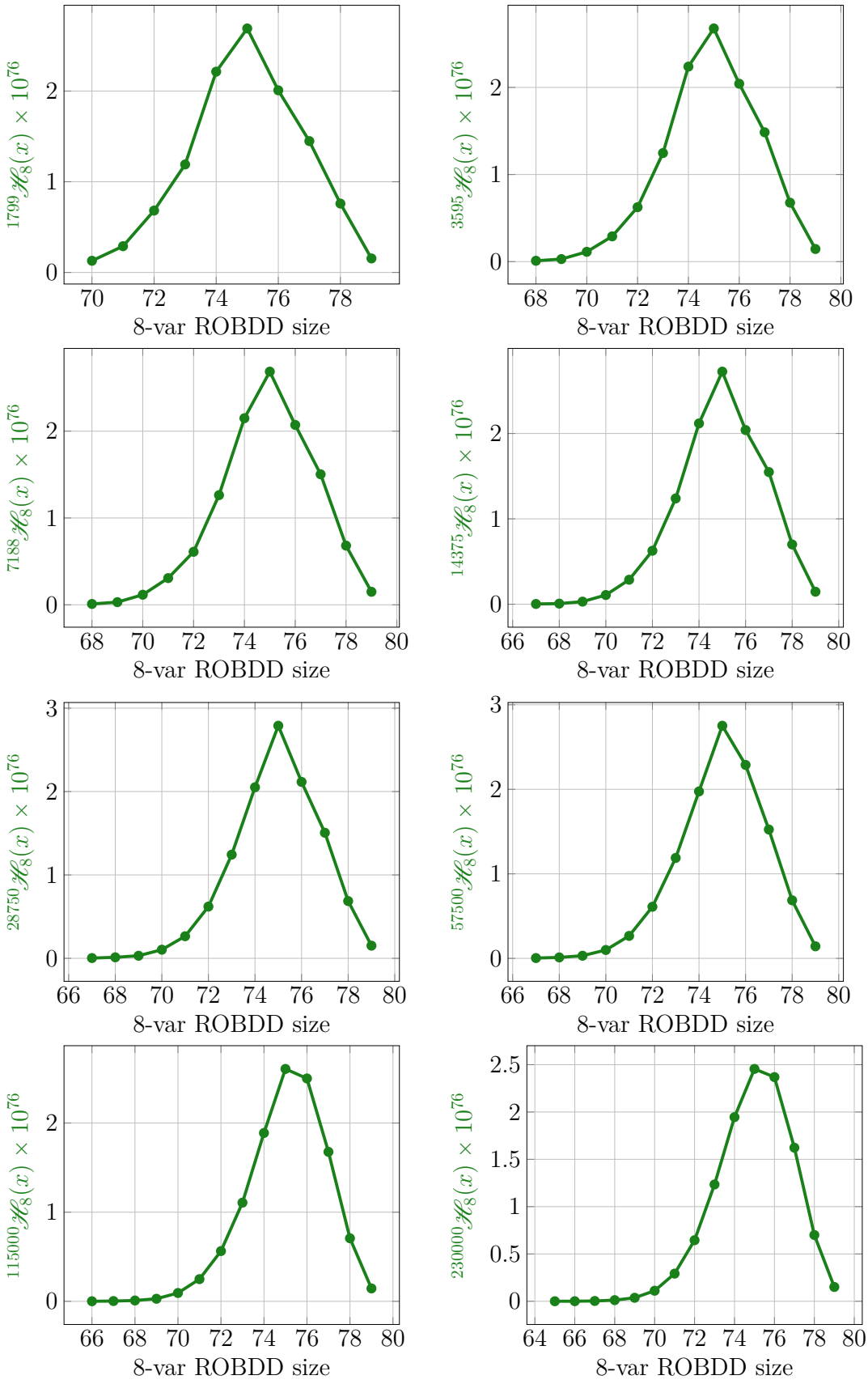


Figure 6.11: Histograms illustrating size distributions of successively larger samples for 8 variables. Each plot shows $y = {}^{M}\mathscr{H}_8$ for successively larger values of M *vs.* x = ROBDD size.

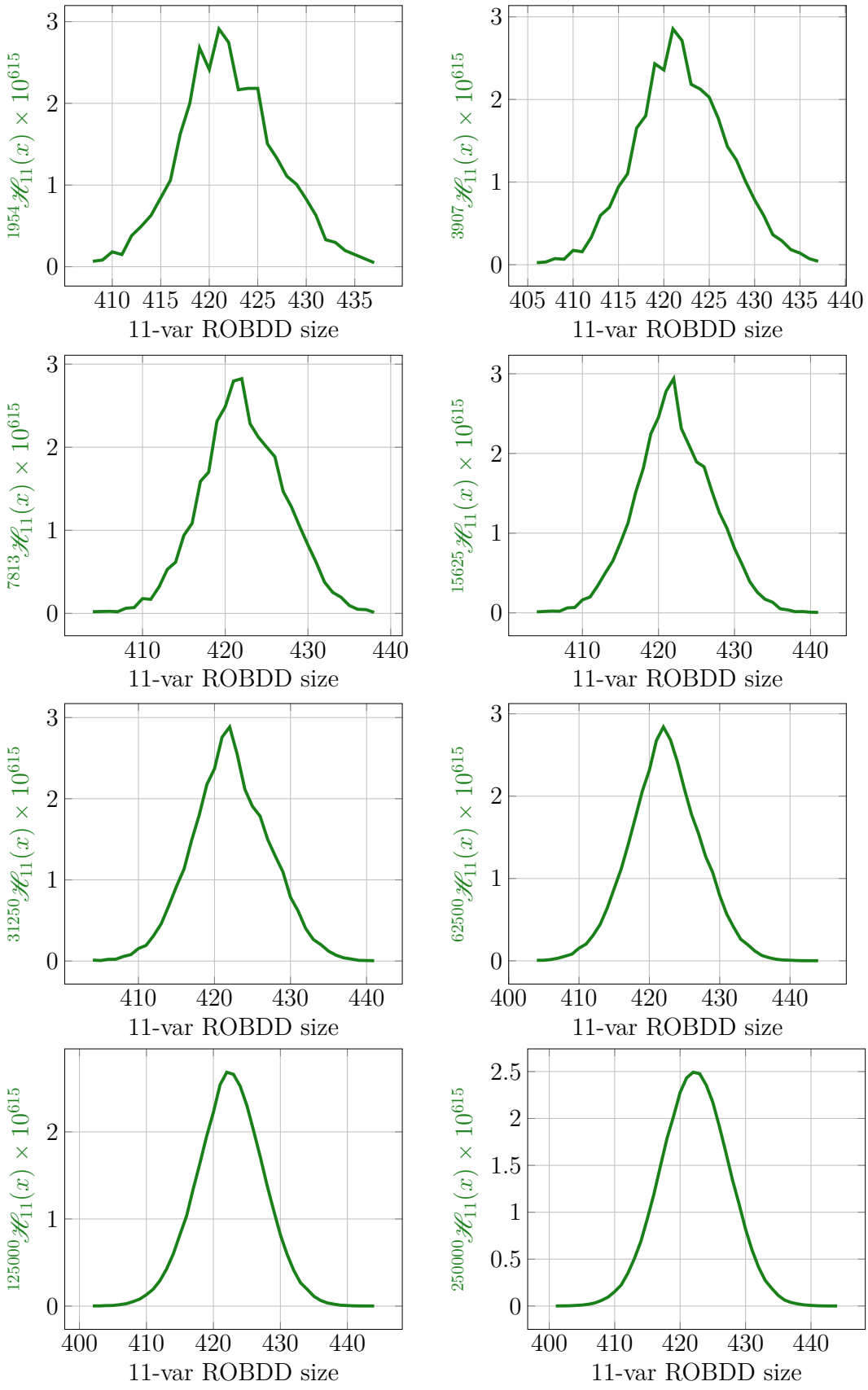


Figure 6.12: Histograms illustrating size distributions of successively larger samples for 11 variables. Each plot shows $y = {}^{M}\mathscr{H}_{11}$ for successively larger values of M *vs.* x = ROBDD size.

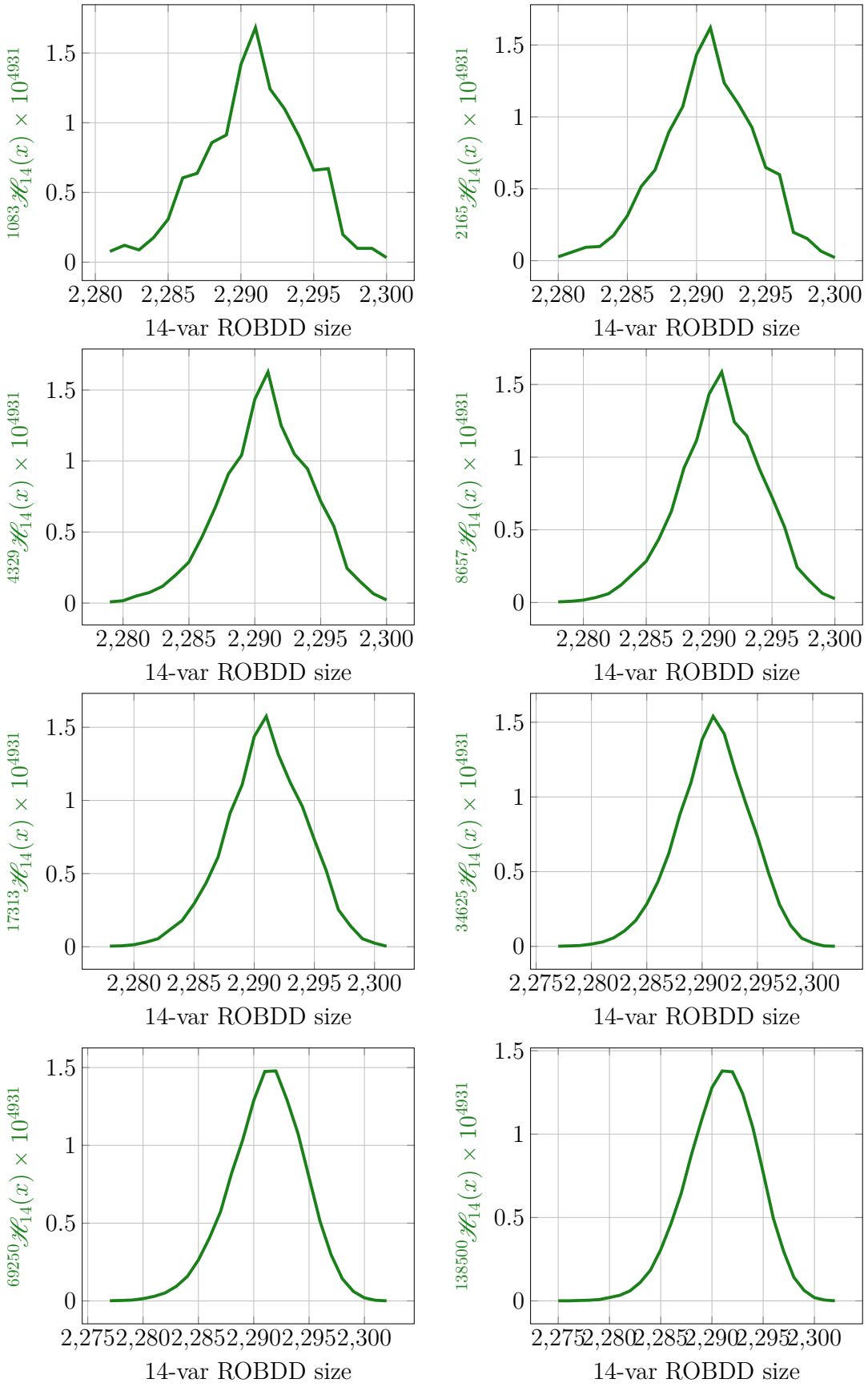


Figure 6.13: Histograms illustrating size distributions of successively larger samples for 14 variables. Each plot shows $y = {}^{M}\mathscr{H}_{14}$ for successively larger values of M *vs.* x = ROBDD size.

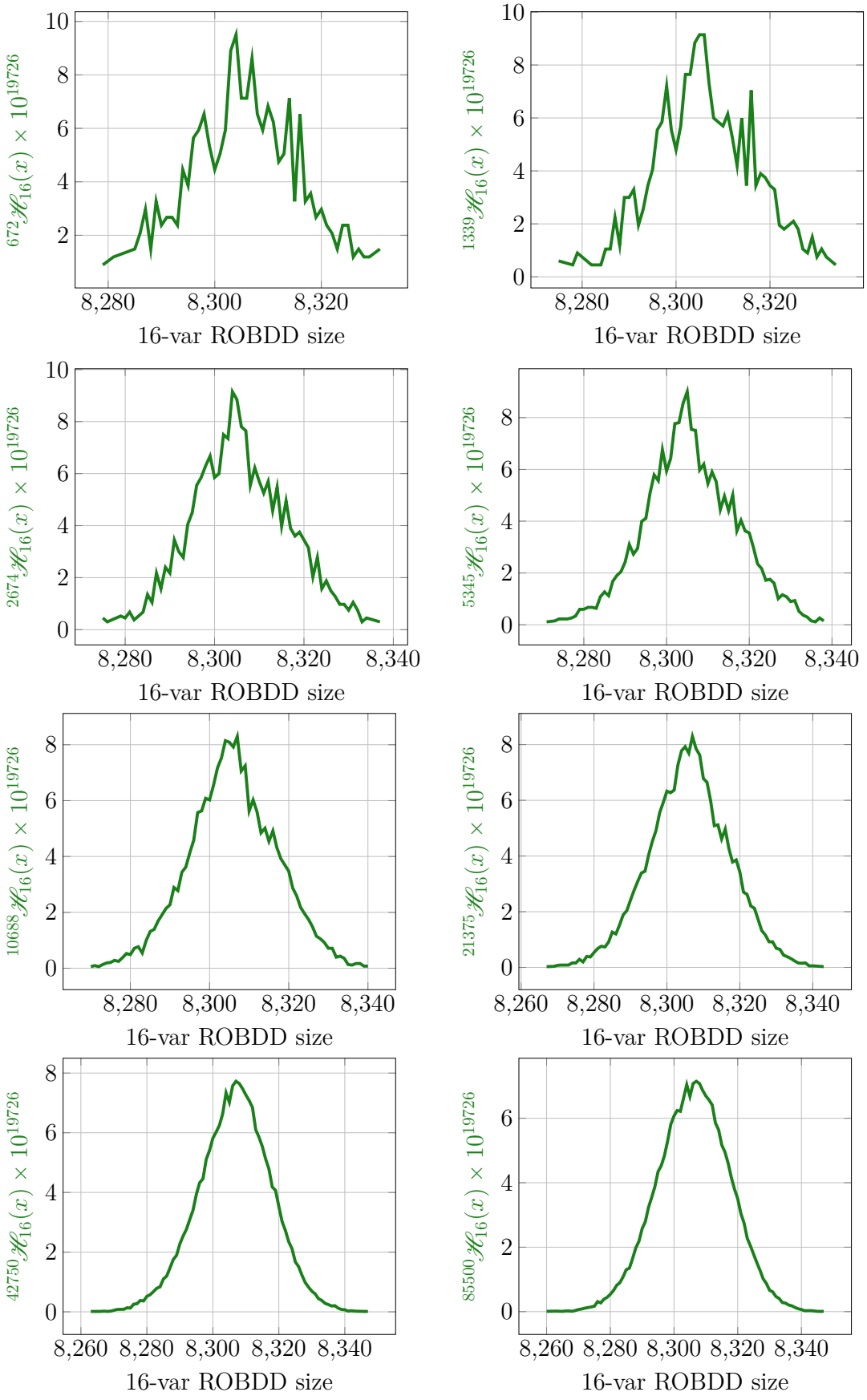


Figure 6.14: Histograms illustrating size distributions of successively larger samples for 16 variables. Each plot shows $y = {}^{M}\mathscr{H}_{16}$ for successively larger values of M *vs.* x = ROBDD size.

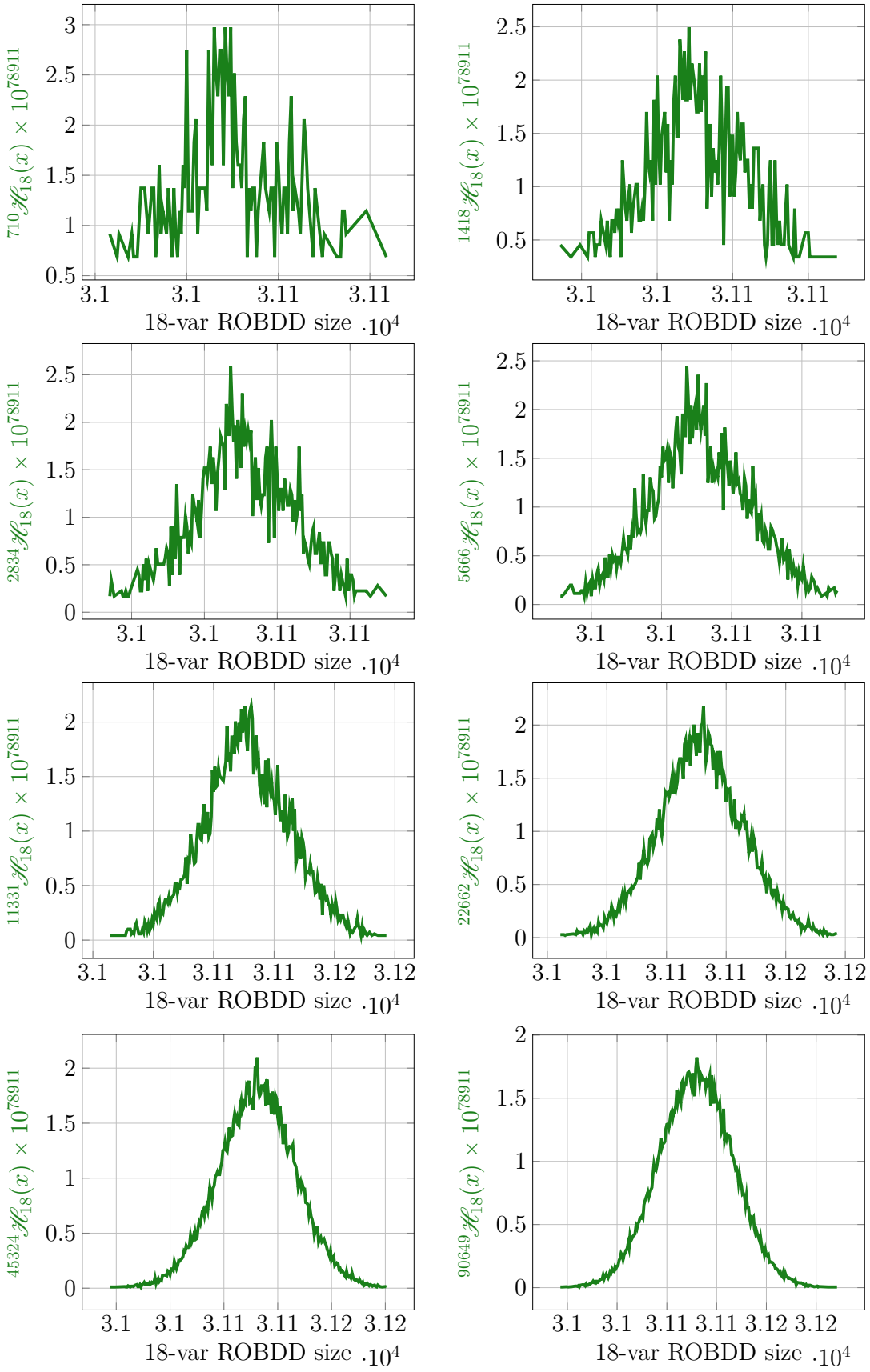


Figure 6.15: Histograms illustrating size distributions of successively larger samples for 18 variables. Each plot shows $y = {}^{M}\mathscr{H}_{18}$ for successively larger values of M *vs.* x = ROBDD size.

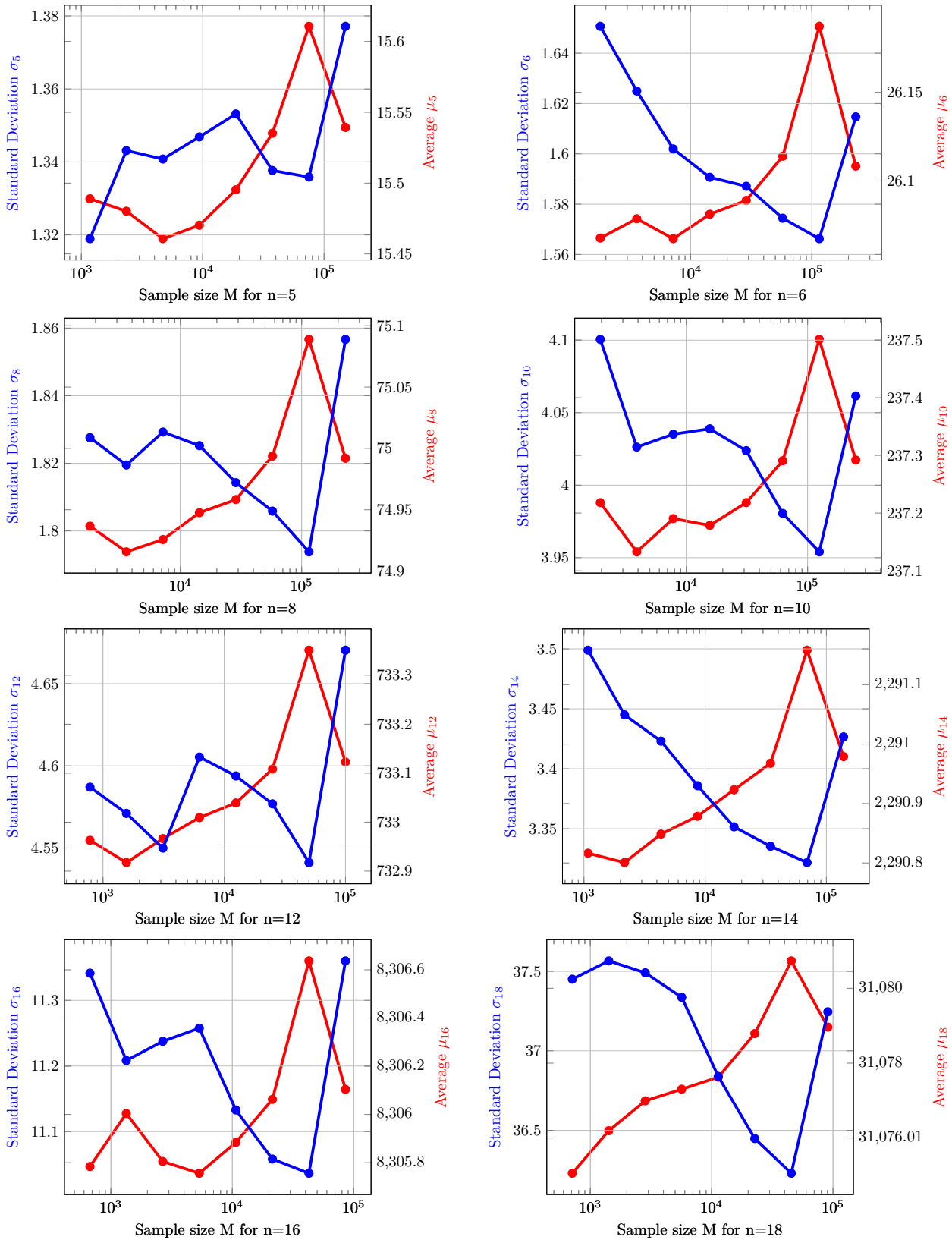


Figure 6.16: Averages (μ_n) and Standard deviations (σ_n) of the 5, 6, 8, 10 ,12, 14, 16, and 18 variable distributions *vs.* sample size, M.

n	$\frac{\mu_{n\,max}-\mu_{n\,min}}{\mu_{n\,final}} \times 100\%$	$\frac{\sigma_{n\,max}-\sigma_{n\,min}}{\sigma_{n\,final}} \times 100\%$
5	$\frac{15.611-15.461}{15.539} = 0.96\%$	$\frac{1.377-1.319}{1.377} = 4.22\%$
6	$\frac{26.187-26.067}{26.108} = 0.46\%$	$\frac{1.651-1.566}{1.615} = 5.23\%$
7	$\frac{44.106-43.987}{44.014} = 0.27\%$	$\frac{1.663-1.610}{1.660} = 3.20\%$
8	$\frac{75.089-74.916}{74.992} = 0.23\%$	$\frac{1.857-1.794}{1.857} = 3.38\%$
9	$\frac{132.189-131.904}{132.040} = 0.22\%$	$\frac{2.743-2.659}{2.743} = 3.05\%$
10	$\frac{237.501-237.133}{237.292} = 0.16\%$	$\frac{4.101-3.954}{4.062} = 3.61\%$
11	$\frac{422.500-422.002}{422.241} = 0.12\%$	$\frac{5.153-5.002}{5.153} = 2.92\%$
12	$\frac{733.351-732.917}{733.123} = 0.06\%$	$\frac{4.670-4.541}{4.670} = 2.77\%$
13	$\frac{1270.825-1270.513}{1270.664} = 0.02\%$	$\frac{2.843-2.754}{2.843} = 3.12\%$
14	$\frac{2291.158-2290.801}{2290.979} = 0.02\%$	$\frac{3.499-3.322}{3.427} = 5.17\%$
15	$\frac{4311.760-4311.308}{4311.482} = 0.01\%$	$\frac{6.135-5.967}{6.135} = 2.75\%$
16	$\frac{8306.637-8305.756}{8306.104} = 0.01\%$	$\frac{11.360-11.037}{11.360} = 2.85\%$
17	$\frac{16118.608-16116.970}{16117.526} = 0.01\%$	$\frac{20.893-20.376}{20.893} = 2.47\%$
18	$\frac{31080.736-31075.063}{31078.967} = 0.02\%$	$\frac{37.566-36.230}{37.246} = 3.59\%$

Figure 6.17: Variation of Average, μ_n, and Standard deviation, σ_n, of histograms for various number of Boolean variables.

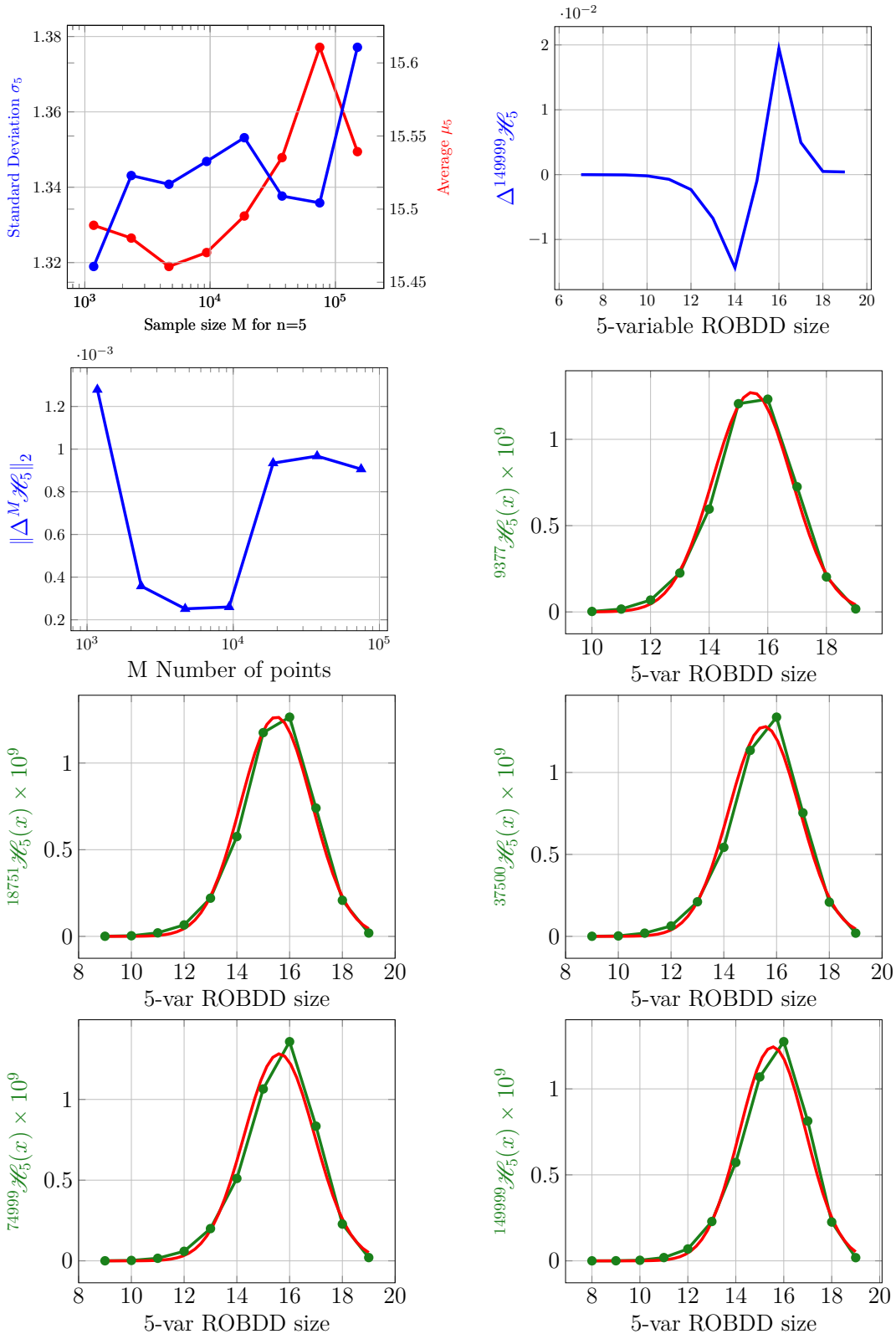


Figure 6.18: Histograms (from Figure 6.10) illustrating size distributions of successively larger samples for 5 variables, shown along with the theoretical normal distribution having the same μ_5 and σ_5. This figure also shows (top left) the plots of $y = \mu_5$ and $y = \sigma_5$ *vs.* $x =$ node count for successively larger values of M. The plot in top right shows the difference function $y = \Delta^M\mathscr{H}_5$ for the maximum sample size taken, M. Also shown in the plot of $y = \|\Delta^M\mathscr{H}_5\|_2$ for successively larger values of M which serves as an indicator of convergence of the $\mathscr{H}_5$ curves.

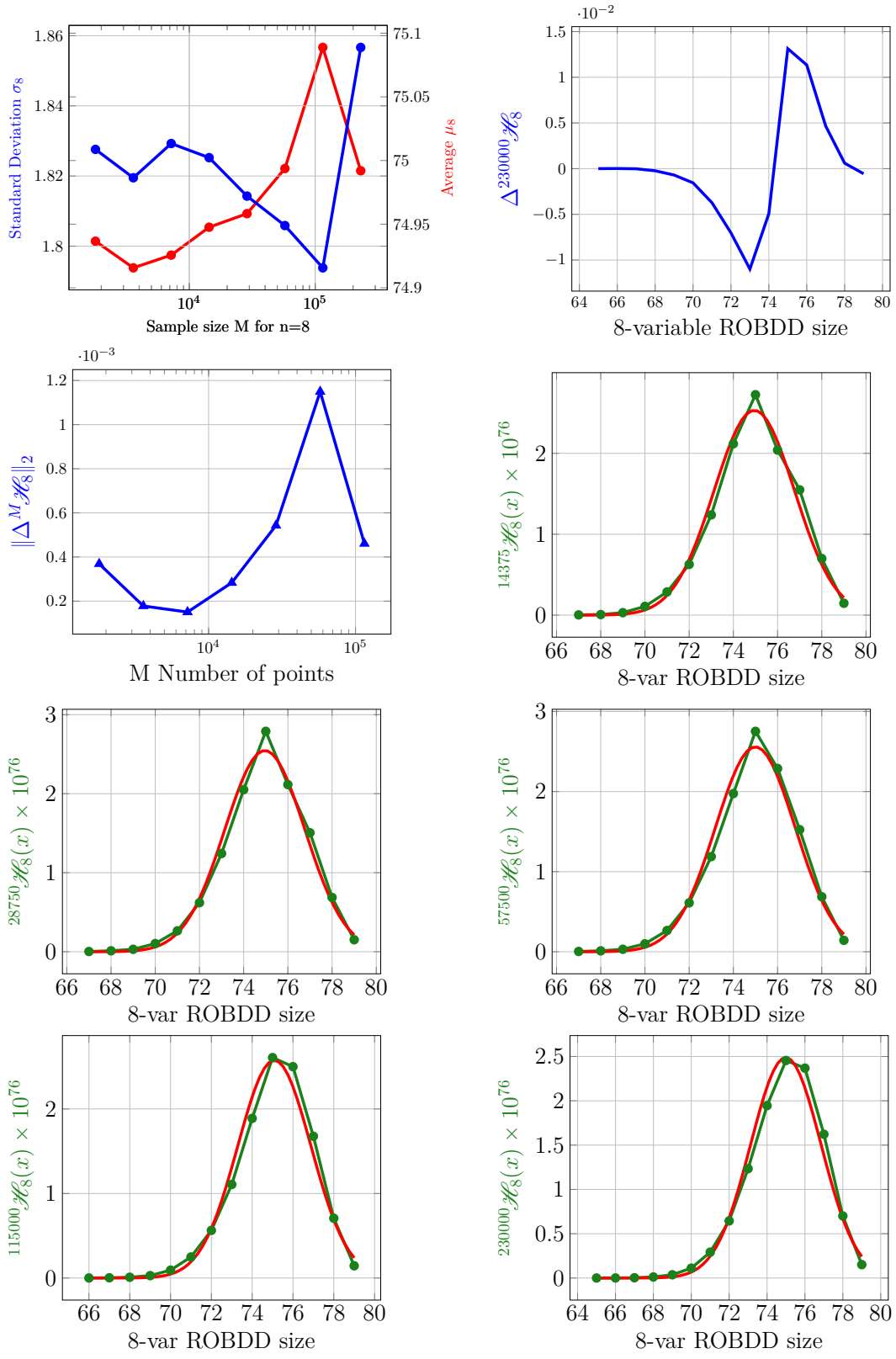


Figure 6.19: Histograms (from Figure 6.11) illustrating size distributions of successively larger samples for 8 variables, shown along with the theoretical normal distribution having the same μ_8 and σ_8. This figure also shows (top left) the plots of $y = \mu_8$ and $y = \sigma_8$ *vs.* $x =$ node count for successively larger values of M. The plot in top right shows the difference function $y = \Delta^M\mathscr{H}_8$ for the maximum sample size taken, M. Also shown in the plot of $y = \|\Delta^M\mathscr{H}_8\|_2$ for successively larger values of M which serves as an indicator of convergence of the $\mathscr{H}_8$ curves.

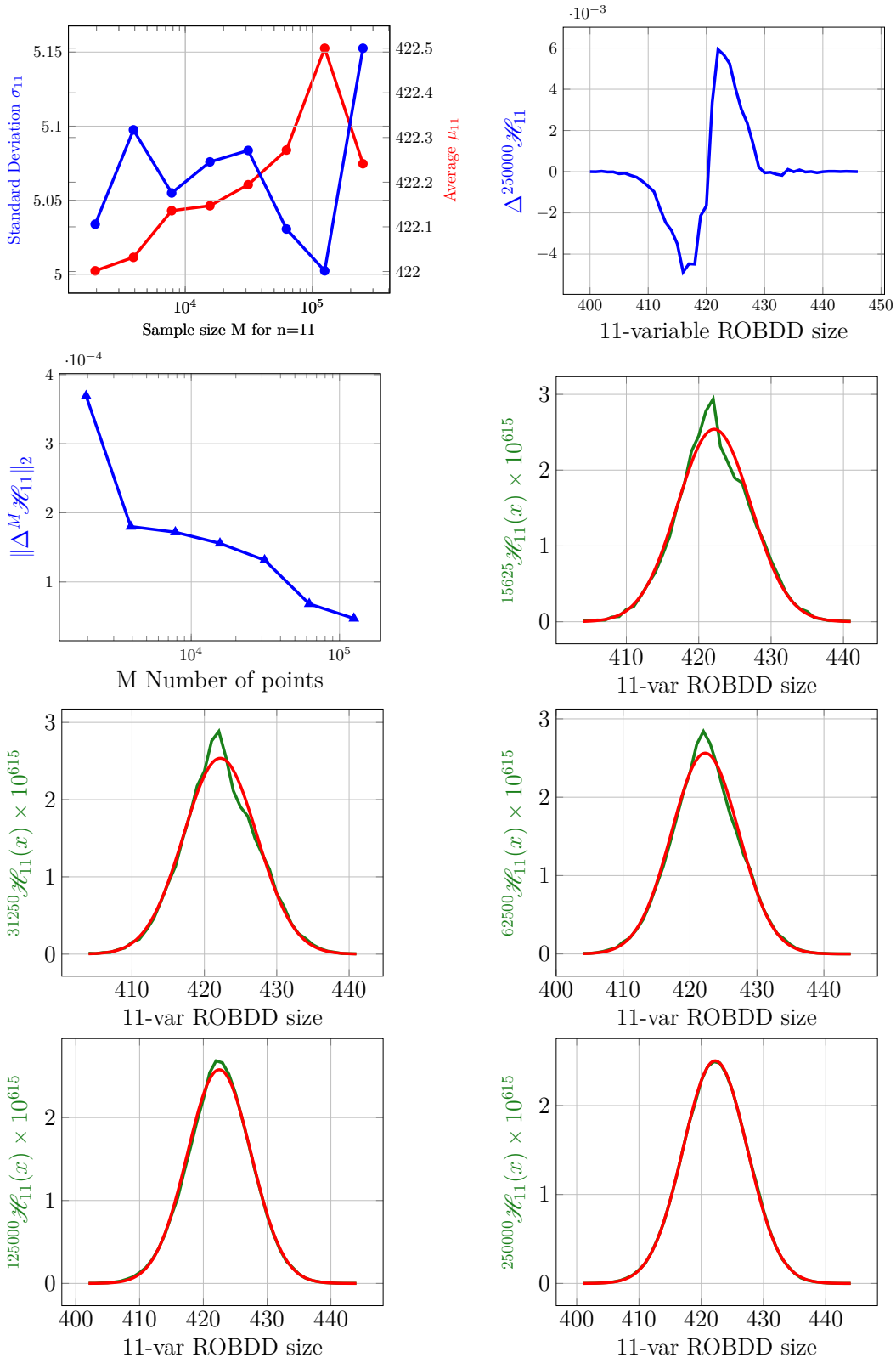


Figure 6.20: Histograms (from Figure 6.12) illustrating size distributions of successively larger samples for 11 variables, shown along with the theoretical normal distribution having the same μ_{11} and σ_{11}. This figure also shows (top left) the plots of $y = \mu_{11}$ and $y = \sigma_{11}$ *vs.* $x =$ node count for successively larger values of M. The plot in top right shows the difference function $y = \Delta^M\mathscr{H}_{11}$ for the maximum sample size taken, M. Also shown in the plot of $y = \|\Delta^M\mathscr{H}_{11}\|_2$ for successively larger values of M which serves as an indicator of convergence of the $\mathscr{H}_{11}$ curves.

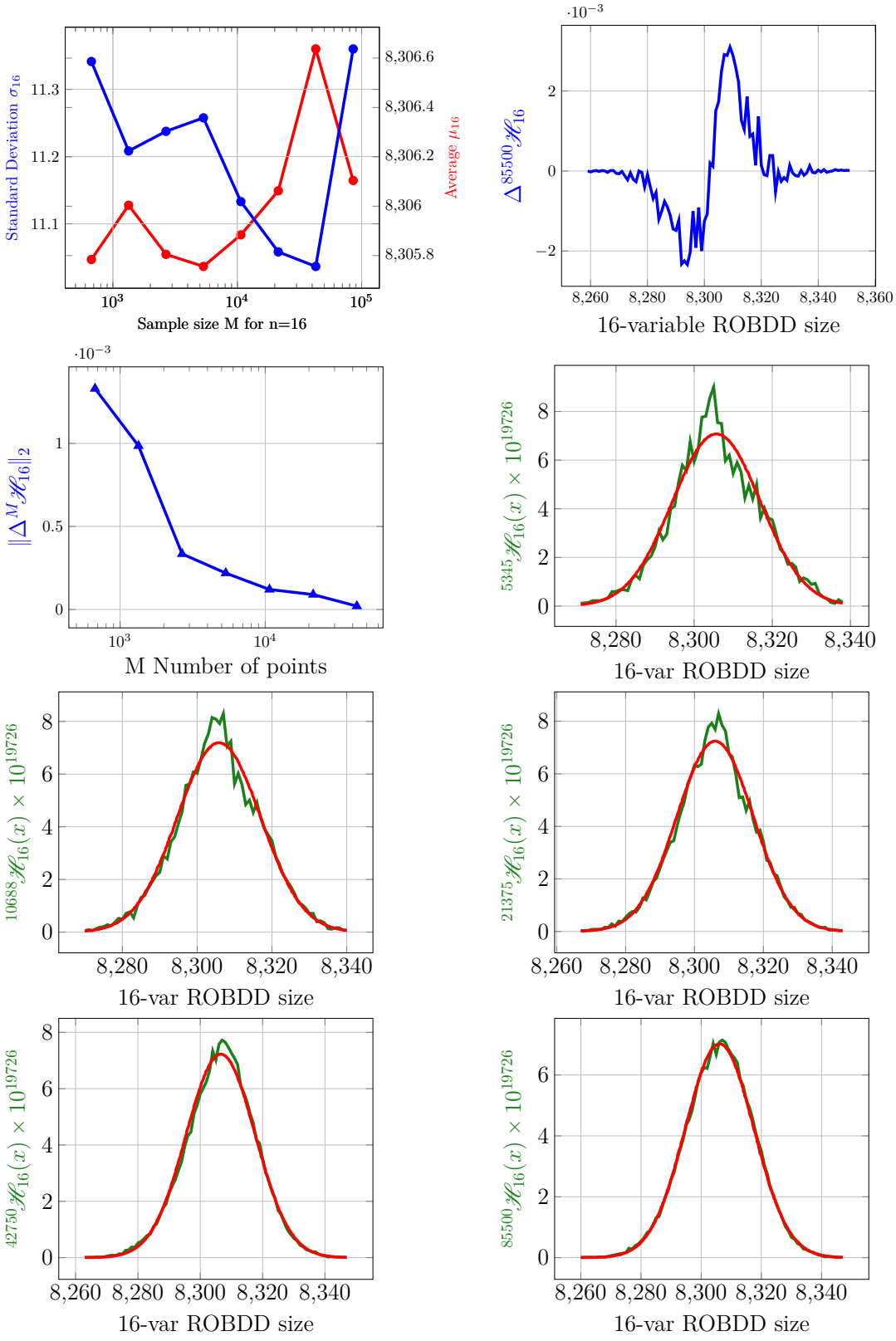


Figure 6.21: Histograms (from Figure 6.14) illustrating size distributions of successively larger samples for 16 variables, shown along with the theoretical normal distribution having the same μ_{16} and σ_{16}. This figure also shows (top left) the plots of $y = \mu_{16}$ and $y = \sigma_{16}$ *vs.* $x =$ node count for successively larger values of M. The plot in top right shows the difference function $y = \Delta^M\mathscr{H}_{16}$ for the maximum sample size taken, M. Also shown in the plot of $y = \|\Delta^M\mathscr{H}_{16}\|_2$ for successively larger values of M which serves as an indicator of convergence of the $\mathscr{H}_{16}$ curves.

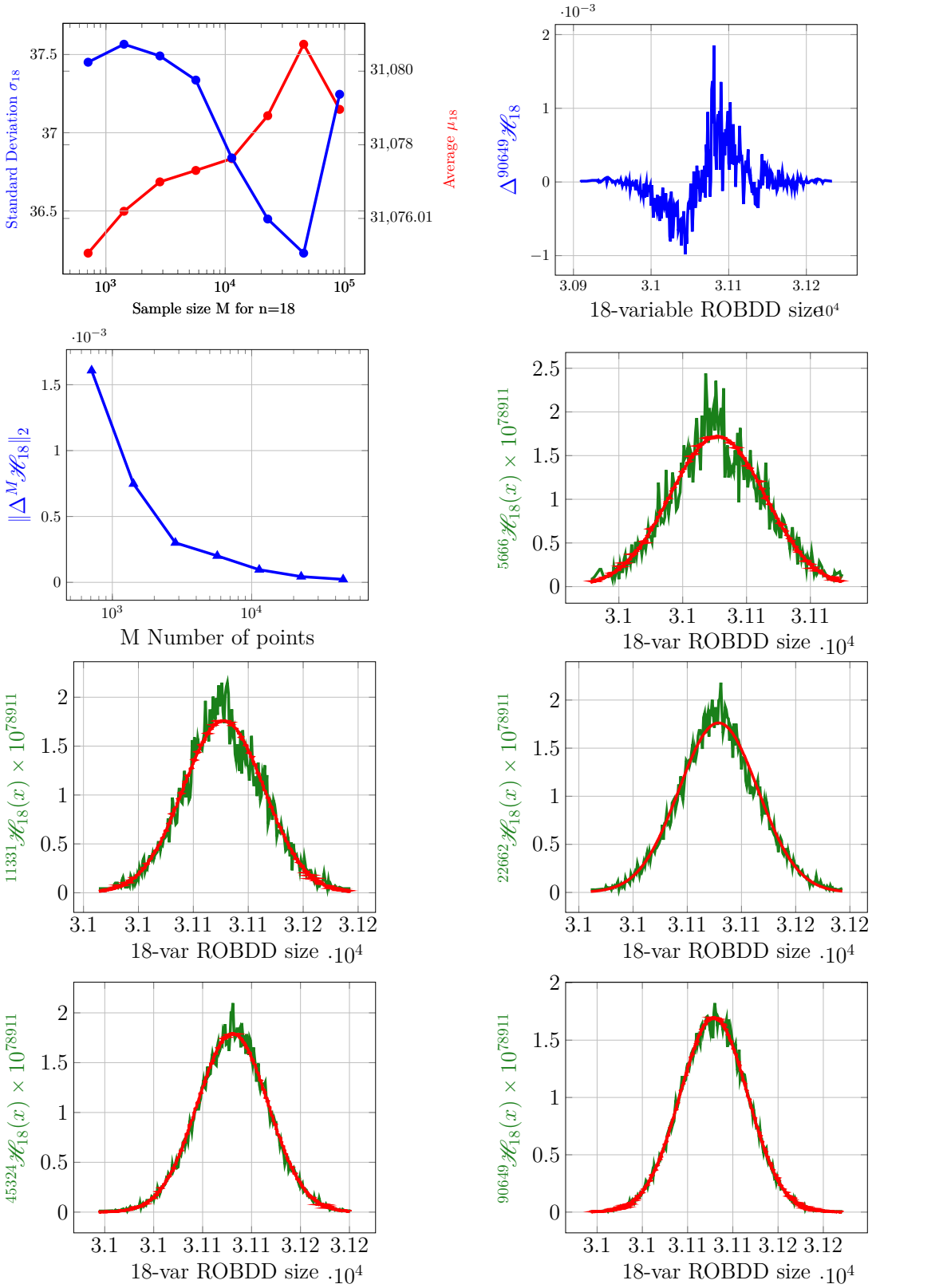


Figure 6.22: Histograms (from Figure 6.15) illustrating size distributions of successively larger samples for 18 variables, shown along with the theoretical normal distribution having the same μ_{18} and σ_{18}. This figure also shows (top left) the plots of $y = \mu_{18}$ and $y = \sigma_{18}$ *vs.* $x =$ node count for successively larger values of M. The plot in top right shows the difference function $y = \Delta^M\mathscr{H}_{18}$ for the maximum sample size taken, M. Also shown in the plot of $y = \|\Delta^M\mathscr{H}_{18}\|_2$ for successively larger values of M which serves as an indicator of convergence of the $\mathscr{H}_{18}$ curves.

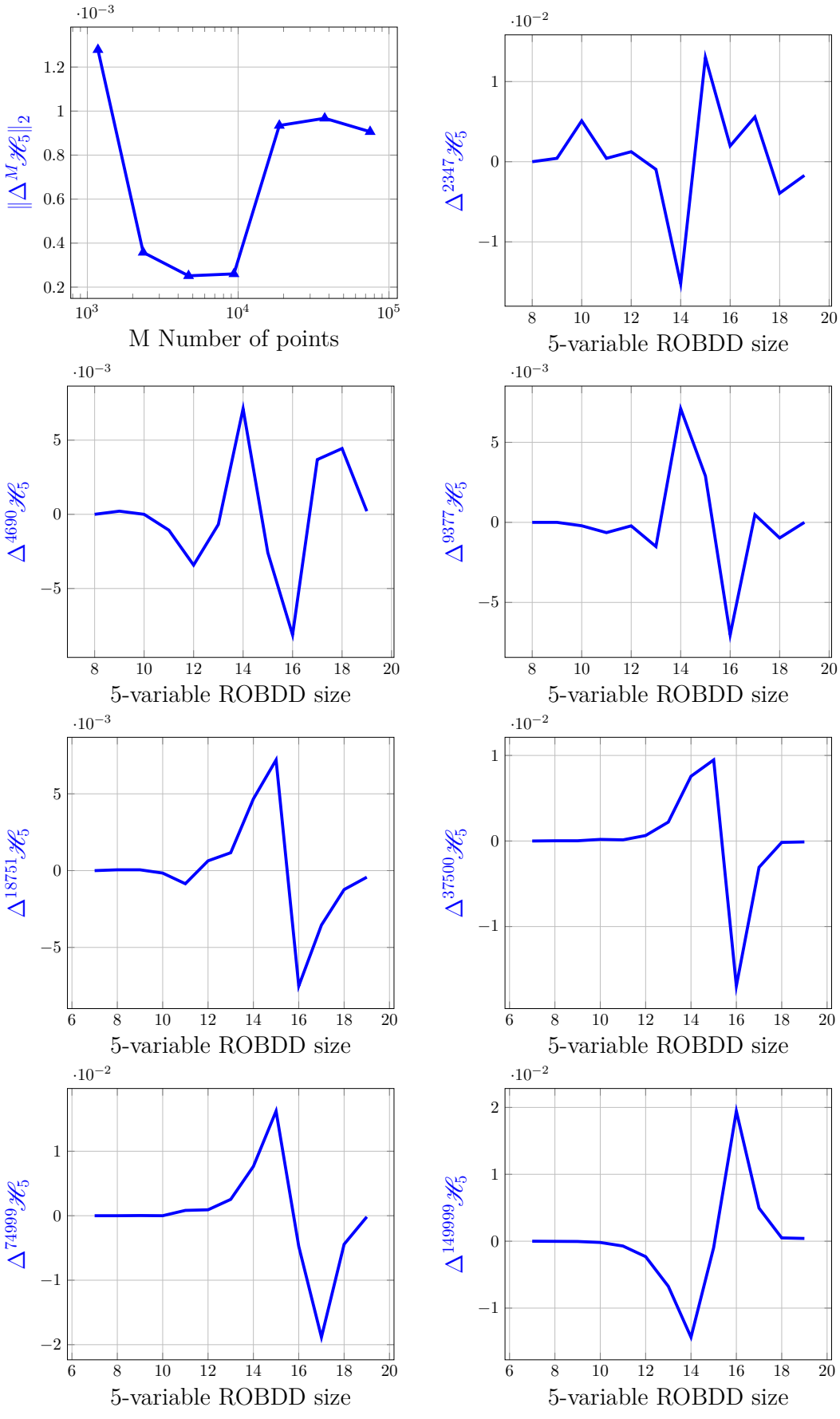


Figure 6.23: Difference functions for $n = 5$

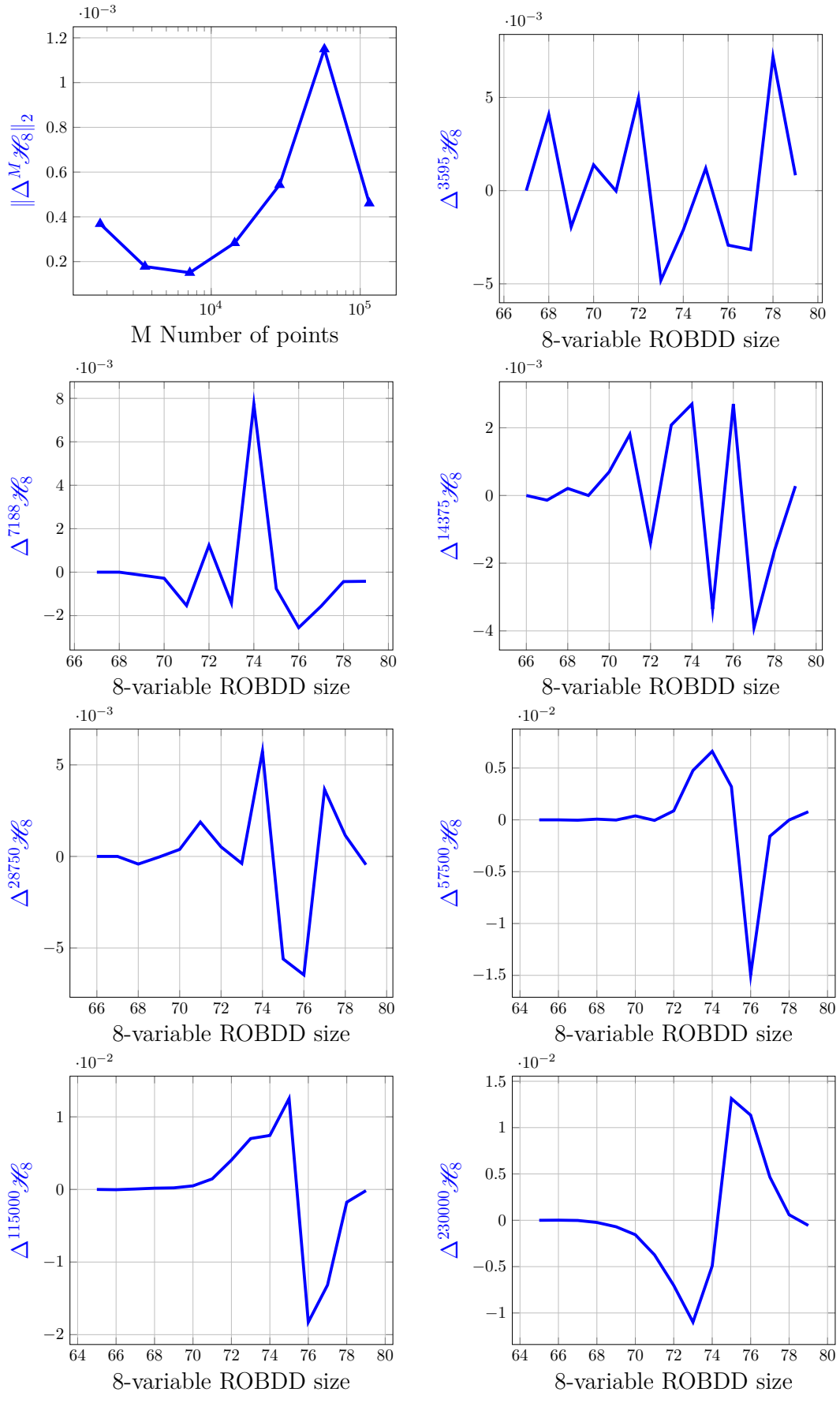


Figure 6.24: Difference functions for $n = 8$

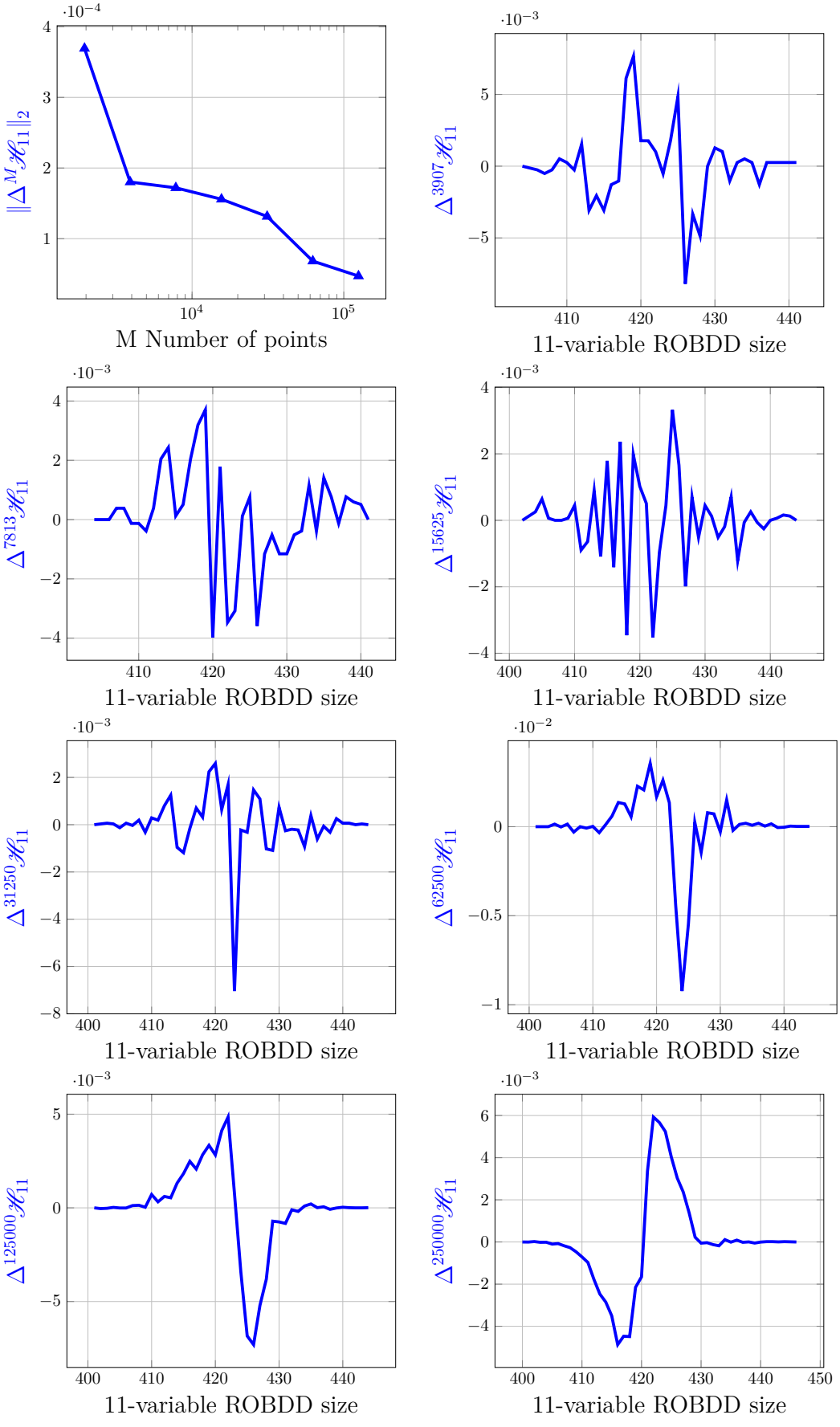


Figure 6.25: Difference functions for $n = 11$

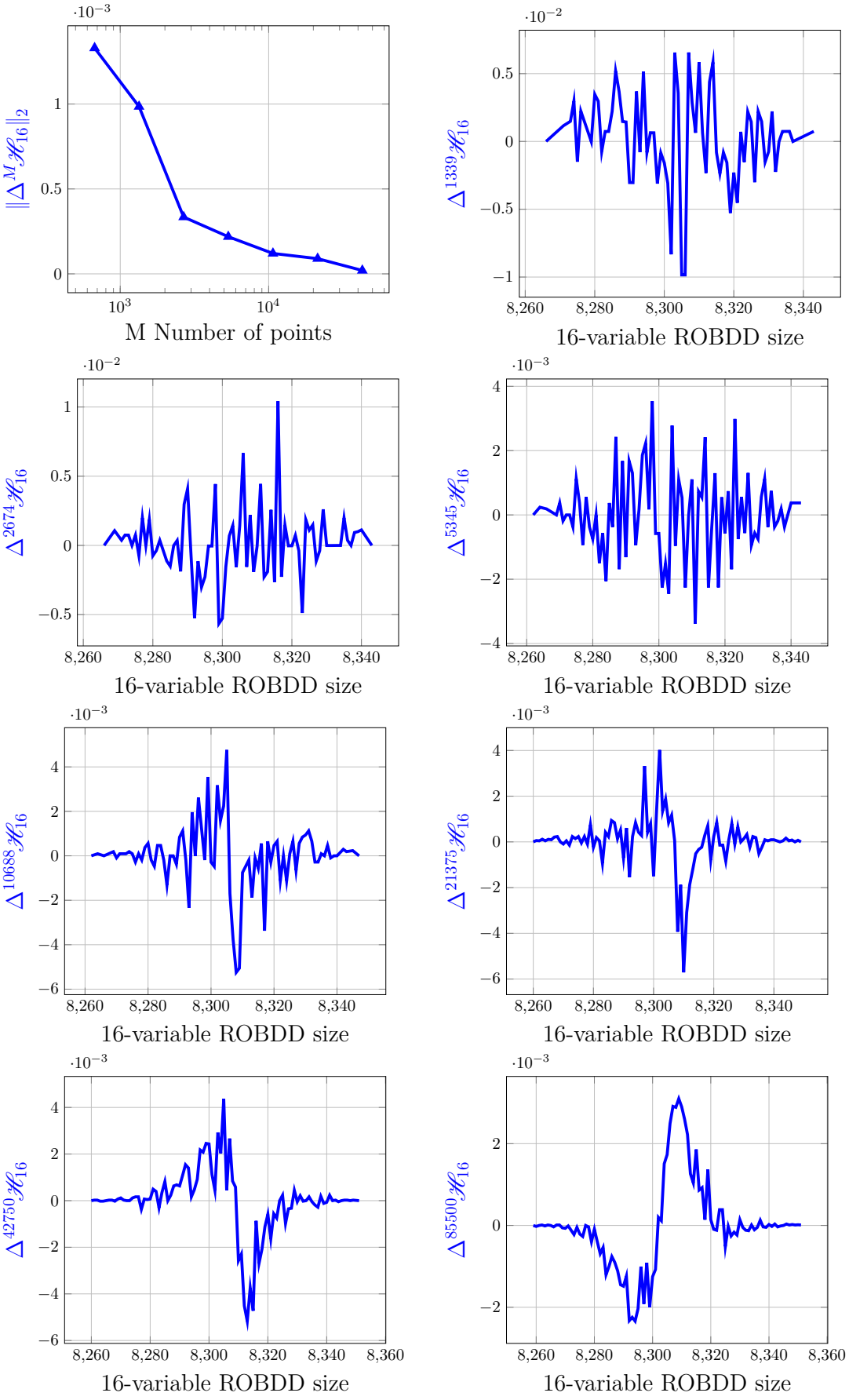


Figure 6.26: Difference functions for $n = 16$

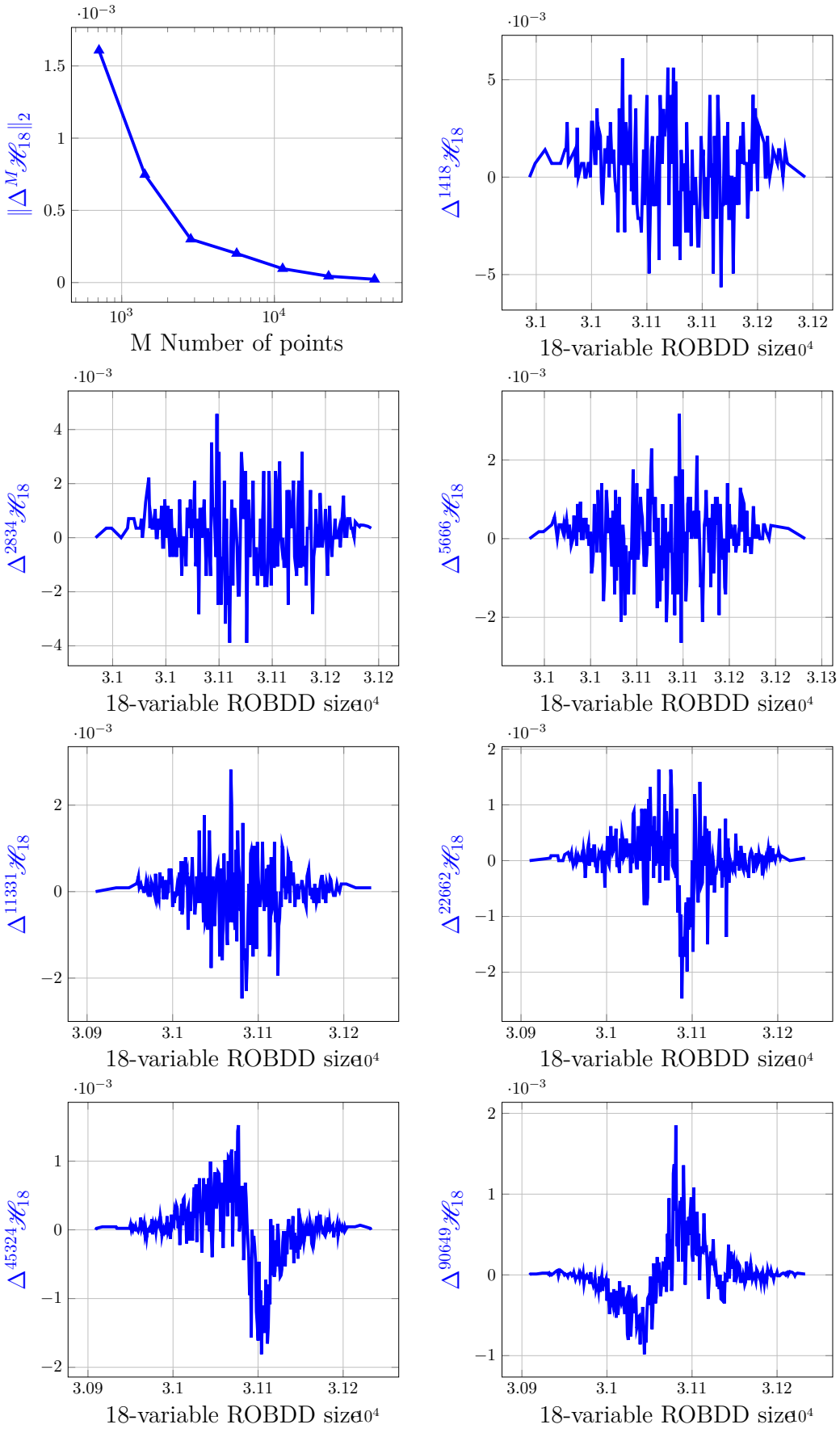


Figure 6.27: Difference functions for $n = 18$

6.1.5 Measuring ROBDD residual compression

In Section 5.1.2, a 31 node UOBDD of 4 Boolean variables was reduced to an equivalent ROBDD with 8 nodes, meaning a residual compression ratio of $8/31 \approx 25.8\%$. The question arises how typical this reduction is? Figure 6.28 shows a plot of the worst-case, average, and median sizes divided by the size of the UOBDD. The figure shows the residual compression ratio, ρ_n, for sizes $n = 1$ through $n = 10$ Boolean variables.

Definition 6.8. The residual compression ratio is defined as

$$\rho_n = \frac{|ROBDD_n|}{|UOBDD_n|}. \tag{6.1}$$

The residual compression ratio quantifies which portion of the original size remains after converting a UOBDD into an ROBDD. The closer to zero, the better the compression.

To calcualte a point in the plot, starting with a number from Figure 6.8 and dividing it by the size of the corresponding UOBDD. A UOBDD of n Boolean variables (as well as a full binary tree of n levels and 2^n leaves) has $|UOBDD_n| = 2^{n+1} - 1$ nodes. It appears from the plot that the residual compression ratio improves (the percentage decreases) as the number of variables increases. It is not clear from the plot what the asymptotic residual compression ratio is, but it appears from experimental data to be less than 15%. It would also appear that whether the residual compression ratio is measured using the average size or worst-case size, the difference becomes negligible as the number of variables increases.

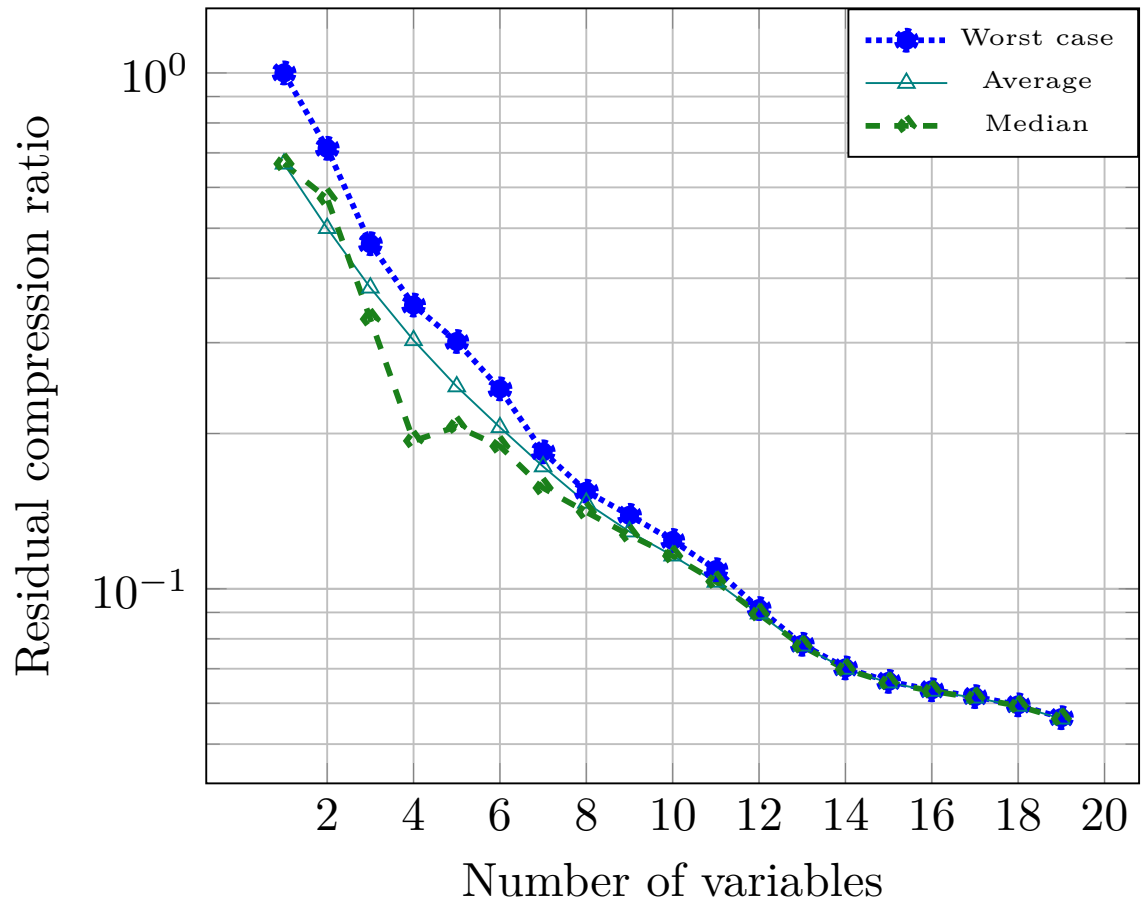


Figure 6.28: Residual compression ratio of ROBDD as compared to UOBDD

In Section 6.1.7, we derive a formula for the worst-case ROBDD size as a function of the number of Boolean variables. In order to do that, we need to understand the topology of such ROBDDs. What are the connectivity invariants which control the shape? In this section, we examine some example worst-case ROBDDs. Section 6.2 discusses an algorithm for constructing such ROBDDs.

Figure 6.29 shows examples of worst-case ROBDD for 1 through 7 variables. Those ROBDDs have 3, 5, 7, 11, 19, 31, and 47 nodes respectively.

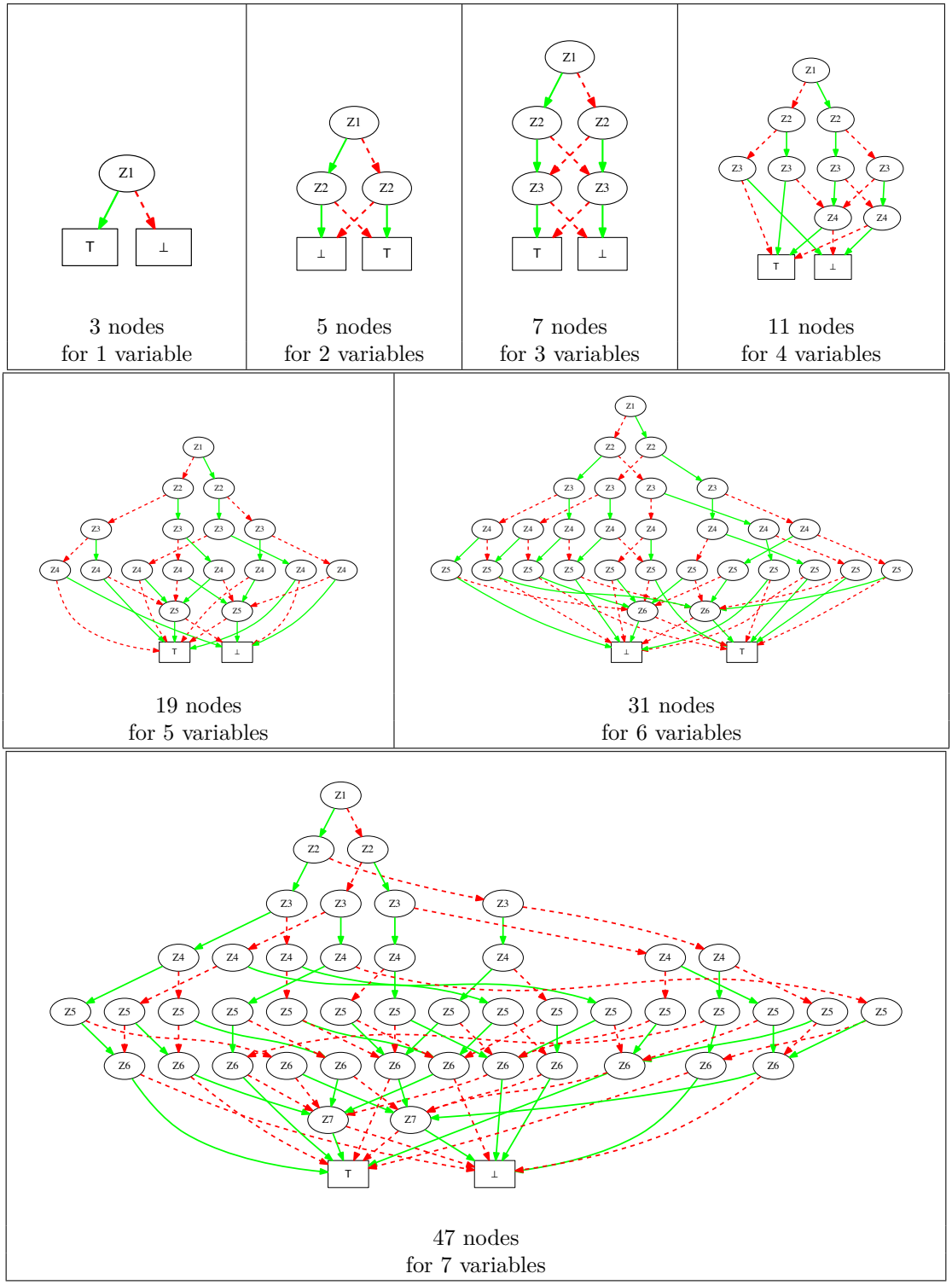


Figure 6.29: Shapes of worst-case ROBDDs for 1 to 7 variables

The 2-variable ROBDD represents the Boolean expression $((Z_1 \wedge \neg Z_2) \vee (\neg Z_1 \wedge Z_2))$, which is the `xor` function. We did not recognize any obvious pattern in the Boolean expressions for the cases of 3 variables or more. As will become clear in Section 6.2 and in Algorithm 4, the worst-case ROBDD is not unique. There is considerable flexibility in constructing it. One may naturally wonder whether there is some underlying pattern within the Boolean expressions corresponding to these worst-case ROBDDs. We have not investigated this question yet, and leave it open for further investigation.

6.1.6 Shape of worst-case ROBDD

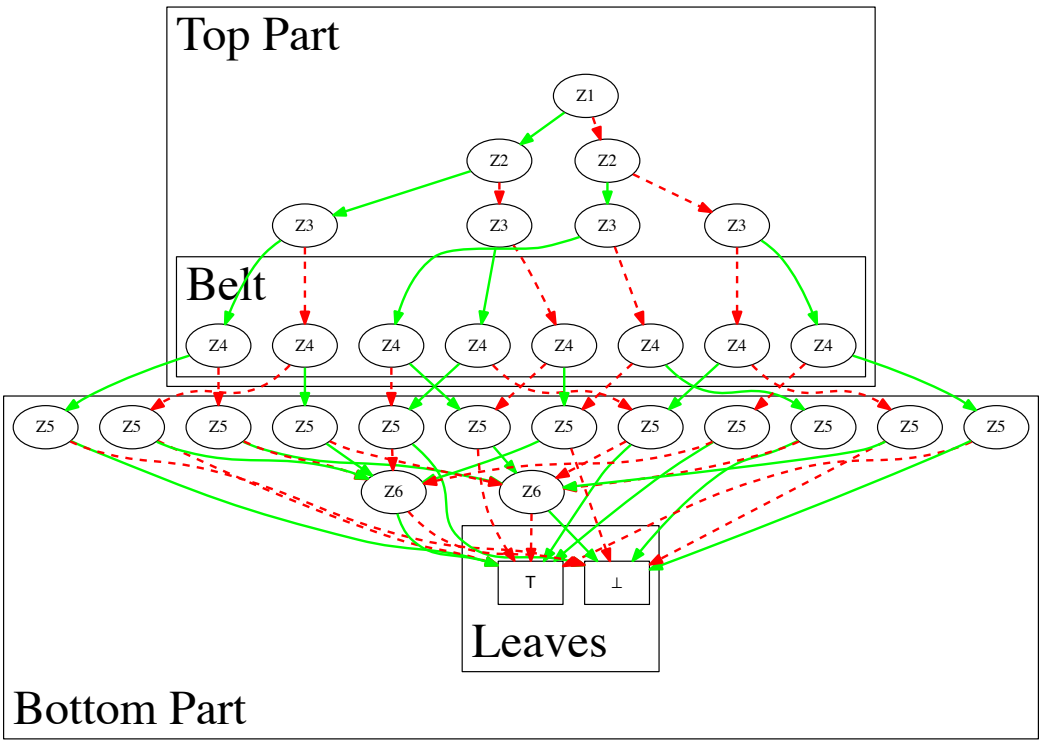


Figure 6.30: Example shape of a worst-case ROBDD. This figure shows 6-variable ROBDD with top & bottom parts marked. The figure also highlights the belt and leaves. Section 6.2 explains the construction of this ROBDD. In particular the leaves, bottom part, belt and top part are specifically constructed in Algorithms 4 and 5. The formula for the row index of the belt is given in Equation (6.29)

Even if there is no obvious pattern among the closed form Boolean expressions, we do notice a general pattern in the overall shapes of the worst-case ROBDDs, as we increase the number of variables. We will make this pattern explicit in Section 6.1.7, but intuitively as shown in Figure 6.30, it seems that the shapes expand from the top (root node) to somewhere close to mid-way down and thereafter contract toward the bottom, always ending with two rows having exactly two nodes each.

This shape makes sense because the maximum possible expansion (top toward bottom) occurs when each row contains twice as many nodes as the row directly above it. Each node in the i^{th} row corresponding to variable Z_i has two arrows (one positive and one negative) pointing to children of variable Z_{i+1}. If the i^{th} row is to have the maximum number of nodes possible, then no node may be symmetric, otherwise the node could be eliminated by the Deletion rule. Furthermore, no two nodes may be congruent, otherwise one of the nodes could be eliminated by the Merging rule.

As shown in Figure 6.30, the top part of the worst-case ROBDD comprises rows which successively double in size. However, this exponential expansion is limited by other factors as seen in Figure 6.30. The expansion terminates at the belt, defined in Definition 6.9.

Definition 6.9. In a worst-case ROBDD the rows from top, downward each have twice the number of nodes as the row above. We define $\mathcal{B}$, the belt, as the maximum index for which this holds. I.e., row $\mathcal{B}$ has twice as many nodes as row $\mathcal{B}-1$, but row $\mathcal{B}+1$ does not.

A formula for $\mathcal{B}$ is given in Equation (6.29) in Section 6.2.

Notice that sometimes, the $\mathcal{B}$ row is the widest row in the ROBDD, such as in the 7-variable case, and other times the $\mathcal{B}+1$ row is the widest row as in the 6-variable case. These two cases are illustrated in Figure 6.29.

One fact that limits the top-down exponential expansion is that the bottommost row must contain exactly two leaf nodes in worst case, corresponding to $\top$, and $\bot$.

Theorem 6.10. *If $m > 1$, the bottommost row of an m-row ROBDD has exactly two nodes.*

Proof. By contradiction. Suppose row had only one of $\top$ or $\bot$. Any node in the second to last row (row $m-1$) would be symmetric, having its positive and negative arrows pointing to this same leaf node in row m. Such a node would be eliminated by the Deletion rule. Row $m-1$ would be empty. Thus, the ROBDD would not have m rows. Contradiction. □

Theorem 6.11. *If $m > 1$, the second to last row, $m-1$ has no more than 2 nodes.*

Proof. We know from Theorem 6.10 that the bottommost row has exactly two leaves. That being the case, if the second to last row had any symmetric node, such a node would be removed by the Deletion rule. Furthermore, if any two nodes in the row were congruent, one of the nodes would be eliminated by the Merging rule. Therefore, as worst case, there may be only as many nodes in the second to last row as there are ordered parings of the leaf nodes. There are only two such ordered pairs: $(\top, \bot)$ and $(\bot, \top)$. The second to last row has no more than two nodes. □

The second to last row is limited to two nodes, by Theorem 6.11. So the worst-case ROBDD has exactly two nodes in the second to last row.

A similar argument limits the third to last row, and the rows above it. In each such case, the number of nodes in the row is limited by the number of ordered pairs which can be formed by all the nodes below it, having no symmetric node and no congruent nodes. This implies a combinatorial expansion from bottom toward the top.

As argued above, there is an exponential growth from the topmost node downward, and there is a combinatorial growth from the bottommost node upward. At some point, the widest part of the graph, these two growth rates meet.

6.1.7 Worst-case ROBDD size

In Section 6.1.2, we saw the worst-case sizes of ROBDDs for different numbers of Boolean variables. We observed an exponential top-down growth rate, a bottom-up combinatorial one, and a point somewhere in between where these two growths meet. In this section, we derive explicit formulas for these observations, and from them, derive the worst-case size $|ROBDD_n|$.

As can be seen in Figure 6.29, the number of nodes per row (per variable), looking from the top-down, is limited by 2^i where i is the index of the variable. The number of nodes per row follows the sequence $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, ...2^k.

The row corresponding to the last variable has two nodes, one with children *positive* $= \bot$, *negative* $= \top$ and one with *positive* $= \top$, *negative* $= \bot$. In the worst case, each row above the bottom has the number of nodes necessary for each node to uniquely connect its positive and negative arrows to some unique pair of nodes below it.

Notation 6.12. The number of ordered pairs of m items is $m^{\underline{2}}$ (read m raised to the second power descending, notice the underscore under the superscript). Recall that $m^{\underline{a}} = \frac{m!}{(m-a)!}$ which, for the special case of $a = 2$, becomes $m^{\underline{2}} = \frac{m!}{(m-2)!} = m \cdot (m-1)$.

$$
\begin{aligned}
{}^nS_n &= {}^nR_{n+1} = 2 & {}^nR_{n+1} &= 2 \\
{}^nS_{n-1} &= {}^nR_{n+1} + {}^nR_n = 4 & {}^nR_n &= {}^nS_n{}^2 = 2 \cdot 1 = 2 \\
{}^nS_{n-2} &= {}^nR_{n+1} + ... + {}^nR_{n-1} = 16 & {}^nR_{n-1} &= {}^nS_{n-1}{}^2 = 4 \cdot 3 = 12 \\
{}^nS_{n-3} &= {}^nR_{n+1} + ... + {}^nR_{n-2} = 256 & {}^nR_{n-2} &= {}^nS_{n-2}{}^2 = 16 \cdot 15 = 240 \\
{}^nS_{n-k} &= \sum_{i=n-(k-1)}^{n+1} {}^nR_i & {}^nR_{n-3} &= {}^nS_{n-3}{}^2 = 256 \cdot 255 = 65280 \\
& & {}^nR_{n-k} &= {}^nS_{n-k}{}^2 \qquad (6.2)
\end{aligned}
$$

Figure 6.31: The two interrelated sequences ${}^nS_i$ and ${}^nR_i$.

Notation 6.13. If $k > \mathcal{B}$, we denote the size of the k^{th} row of the worst-case ROBDD of n variables as ${}^nR_k$

Definition 6.14. If $k > \mathcal{B}$, we define ${}^nS_k$ to be the total number of nodes of rows $k+1$ through n. In other words, ${}^nS_k$ is the number of nodes in the rows strictly below row k.

$$
{}^nS_{n-k} = \sum_{i=n-(k-1)}^{n+1} {}^nR_i
$$

Viewed from the bottom-up, the sequence of rows have sizes ${}^nR_{n-1}, {}^nR_{n-2}, {}^nR_{n-3}$, *etc.*. The number of nodes in row i is a function of the sum of the number of nodes in the rows below it, namely ${}^nR_i = {}^nS_i{}^2 = {}^nS_i \cdot ({}^nS_i - 1)$.

Notice that the bottom row of a non-trivial worst-case ROBDD has exactly 2 nodes, the $\top$ and $\bot$ nodes, thus ${}^nR_{n+1} = 2$. For each i, ${}^nS_i$ can be calculated as the sum of the previous ${}^nR_j$ for $j = n-i, ..., n+1$. These progressions are illustrated by the equations in Figure 6.31.

An interesting pattern emerges: ${}^nS_n = 2^{2^0}$, ${}^nS_{n-1} = 2^{2^1}$, ${}^nS_2 = 2^{2^2}$, ${}^nS_3 = 2^{2^3}$, suggesting Lemma 6.15.

Lemma 6.15. *Let*

$$
{}^nS_{n-k} = \sum_{i=n-(k-1)}^{n+1} {}^nR_i \,,
$$

where

$$
{}^nR_{n+1} = 2
$$

and for $k > 1$,

$$
{}^nR_{n-k} = {}^nS_{n-k} \cdot ({}^nS_{n-k} - 1) \,.
$$

Then for every positive integer k,

$$
{}^nS_{n-k} = 2^{2^k} \,.
$$

Proof. By Induction: The initial case, $k = 0$ is that

$$
\begin{aligned}
{}^nS_{n-k} &= {}^nS_{n-0} \\
&= {}^nR_{n+1} = 2 = 2^1 = 2^{2^0} = 2^{2^k} \,.
\end{aligned}
$$

The only thing that remains is to be shown that for $k \geqslant 0$, ${}^nS_{n-k} = 2^{2^k}$ implies ${}^nS_{n-(k+1)} = 2^{2^{k+1}}$. Assume

$$ {}^nS_{n-k} = 2^{2^k} \, . $$

It follows that

$$
\begin{aligned}
{}^nS_{n-(k+1)} &= \sum_{i=n-k}^{n+1} {}^nR_i \\
&= {}^nR_{n-k} + \sum_{i=n-(k+1)}^{n+1} {}^nR_i \\
&= {}^nR_{n-k} + {}^nS_{n-k} \\
&= ({}^nS_{n-k}) \cdot ({}^nS_{n-k} - 1) + ({}^nS_{n-k}) \\
&= ({}^nS_{n-k}) \cdot ({}^nS_{n-k} - 1 + 1) \\
&= {}^nS_{n-k} \cdot {}^nS_{n-k} \\
&= ({}^nS_{n-k})^2 \\
&= (2^{2^k})^2 \quad = \quad 2^{2 \cdot 2^k} \quad = \quad 2^{2^{k+1}} \, .
\end{aligned}
$$

□

Next, we show more concise forms for ${}^nR_{n-k}$ and ${}^nR_i$. As a convention, we will use the variable i to index rows and summations, when counting from the top (root) node down. By contrast, we will use the variable k to index rows and summations, when counting from the bottom-up.

Lemma 6.16. *If $k \geqslant 0$, then*

$$ {}^nR_{n-k} = 2^{2^{k+1}} - 2^{2^k} \, , $$

and if $i \leqslant n$,

$$ {}^nR_i = 2^{2^{n-i+1}} - 2^{2^{n-i}} \, . $$

Proof.

$$
\begin{aligned}
{}^nR_{n-k} &= {}^nS_{n-k}{}^2 && \text{by 6.2} \\
&= (2^{2^k}) \cdot (2^{2^k} - 1) && \text{by Lemma 6.15} \\
&= (2^{2^k} \cdot 2^{2^k}) - 2^{2^k} \\
&= (2^{2^k})^2 - 2^{2^k} \\
&= 2^{2 \cdot 2^k} - 2^{2^k} \\
{}^nR_{n-k} &= \begin{cases} 2^{2^{k+1}} - 2^{2^k} & \text{if } k \geqslant 0 \\ 2 & \text{if } k = -1 \end{cases} \\
{}^nR_i &= \begin{cases} 2^{2^{n-i+1}} - 2^{2^{n-i}} & \text{if } i \leqslant n \\ 2 & \text{if } i = n+1 \end{cases}
\end{aligned}
$$

□

Notation 6.17. If $i \leqslant \mathcal{B}$, we denote the size of the i^{th} row of the worst-case ROBDD of n variables as ${}^nr_i$.

As explained already, ${}^nR_i$ is the number of elements which would fit into row i, only taking into consideration the combinatorial growth from the bottommost row up to row i. However, when we look from the topmost row down, and we take only the exponential growth into account, we see the number of nodes in row i is given by

$$ {}^{n}r_i = 2^{i-1} \tag{6.3}$$

$$ {}^{n}r_{n-k} = 2^{n-k-1}\,. \tag{6.4}$$

Each row within the worst-case ROBDD is limited in size by the two terms, ${}^{n}R_i$ and ${}^{n}r_i$. The precise number of nodes in each row is the minimum of these two terms. The total number of nodes in a worst-case ROBDD of n variables is the sum of the number of nodes in each of its rows, given by Equation 6.6 which holds when $n > 1$.

$$ |ROBDD_n| = 2 + \sum_{i=1}^{n} \min\{{}^{n}r_i, {}^{n}R_i\} \tag{6.5}$$

$$ = 2 + \sum_{i=1}^{n} \min\{2^{i-1}, 2^{2^{n-i+1}} - 2^{2^{n-i}}\} \tag{6.6}$$

Theorem 6.18 is stated and proven now. This theorem is useful in the later discussion of Algorithm 4.

Theorem 6.18. *Every row of a worst-case ROBDD, except the first row, has an even number of nodes.*

Proof. The i'th row of an n-variable ROBDD has either ${}^{n}r_i$ nodes or ${}^{n}R_i$ nodes. If $i > 1$, then ${}^{n}r_i = 2^{i-1}$ (by Equation 6.3) is even. If $1 < i \leqslant n$, then ${}^{n}R_i = 2^{2^{n-i+1}} - 2^{2^{n-i}}$ (by Lemma 6.16) is even. The final case is when $i = n + 1$, the row of terminal nodes, ${}^{n}R_{n+1} = 2$ which is even. □

In Section 6.1.7 we derived the sizes of the rows of the worst-case ROBDD. We also derived Equation 6.6 which is easy to state but difficult to evaluate, which gives the value of $|ROBDD_n|$ in terms of the sum of the row sizes. In this section we will build up to and prove Theorem 6.24 which makes a step forward in making this value easier to calculate.

Corollary 6.19. *A worst-case ROBDD has an odd number of nodes.*

We could prove Corollary 6.19 by straightforward examination of Equation 6.6, and we would reach the same conclusion as the following simpler argument.

Proof. The first row of any ROBDD (worst-case or not) has a single node. By Theorem 6.18 every row thereafter of a worst-case ROBDD has an even number of nodes. Therefore, the total number of nodes is necessarily odd. □

There is a shortcut for calculating the sum in Equation 6.6. To make the shortcut more evident, first consider the example where $n = 10$, and calculate the size of row $i = 1$. To calculate $\min\{2^{1-1}, 2^{2^{10-1+1}} - 2^{2^{10-1}}\} = \min\{2^0, 2^{2^{10}} - 2^{2^9}\} = 1$, there is no need to calculate the 1024 digits of $2^{2^{10}} - 2^{2^9}$, because $2^0 = 1$ is obviously smaller. The trick is to realize that the summation (Equation 6.6) is the sum of leading terms of the form 2^i, plus the sum of trailing terms of the form $2^{2^{n-i+1}} - 2^{2^{n-i}}$. How many leading and trailing terms may not be obvious however. Theorem 6.24 shows the existence of the so-called threshold function, θ, which makes these two sums explicit.

6.1.8 The threshold function θ

In this section, we prove the existence of the so-called threshold function, θ, and express $|ROBDD_n|$ in terms of θ (Theorem 6.24). Before proving that lemma, we establish a few intermediate results to simplify later calculations.

Lemma 6.20. *If $f : \mathbb{R} \to \mathbb{R}$ is differentiable, then*

$$ \frac{d}{dx} 2^{f(x)} = 2^{f(x)} \cdot \ln 2 \cdot \frac{d}{dx} f(x) $$

Proof.

$$\begin{aligned}\frac{d}{dx}2^{f(x)} &= \frac{d}{dx}e^{\ln 2^{f(x)}} = \frac{d}{dx}e^{f(x)\cdot\ln 2}\\ &= e^{f(x)\cdot\ln 2}\cdot\ln 2\cdot\frac{d}{dx}f(x)\\ &= e^{\ln 2^{f(x)}}\cdot\ln 2\cdot\frac{d}{dx}f(x)\\ &= 2^{f(x)}\cdot\ln 2\cdot\frac{d}{dx}f(x)\end{aligned}$$

□

Lemma 6.21. *If $f:\mathbb{R}\to\mathbb{R}$ is differentiable, then*

$$\frac{d}{dx}2^{2^{f(x)}} = 2^{f(x)+2^{f(x)}}\cdot\ln 4\cdot\frac{d}{dx}f(x)$$

Proof. Change of variables, let $g(x)=2^{f(x)}$, and use Lemma 6.20 twice.

$$\begin{aligned}\frac{d}{dx}2^{2^{f(x)}} &= \frac{d}{dx}2^{g(x)} = 2^{g(x)}\cdot\ln 2\cdot\frac{d}{dx}g(x)\\ &= \frac{d}{dx}2^{f(x)}\cdot\ln 2\cdot 2^{2^{f(x)}}\\ &= \left(2^{f(x)}\cdot\ln 2\cdot\frac{d}{dx}f(x)\right)\cdot\left(\ln 2\cdot 2^{2^{f(x)}}\right)\\ &= 2^{f(x)+2^{f(x)}}\cdot\ln 4\cdot\frac{d}{dx}f(x)\end{aligned}$$

□

Even though Lemma 6.22 is trivial to prove, we provide it because it removes redundant steps in proving Lemmas 6.23 and 6.29.

Lemma 6.22. *If $h:\mathbb{R}\to\mathbb{R}$, then $2^{h(x)+1+2^{h(x)+1}} > 2^{h(x)+2^{h(x)}}$.*

Proof.

$$\begin{aligned}h(x)+1 > h(x) &\implies 2^{h(x)+1} > 2^{h(x)}\\ &\implies h(x)+1+2^{h(x)+1} > h(x)+2^{h(x)}\\ &\implies 2^{h(x)+1+2^{h(x)+1}} > 2^{h(x)+2^{h(x)}}\end{aligned}$$

□

Lemma 6.23. *The function, $f(x)=2^{2^{n-x+1}}-2^{2^{n-x}}$ is decreasing.*

Proof. To show that f is decreasing, we show that $\frac{d}{dx}f(x)<0$.

$$\begin{aligned}\frac{d}{dx}f(x) &= \frac{d}{dx}\left(2^{2^{n-x+1}}-2^{2^{n-x}}\right)\\ &= 2^{n-x+1+2^{n-x+1}}\cdot\ln 4\cdot(-1) - 2^{n-x+2^{n-x}}\cdot\ln 4\cdot(-1) && \text{by Lemma 6.21}\\ &= \left(2^{n-x+1+2^{n-x+1}} - 2^{n-x+2^{n-x}}\right)\cdot\ln 4\cdot(-1)\\ &= \left(2^{n-x+2^{n-x}} - 2^{n-x+1+2^{n-x+1}}\right)\cdot\ln 4\end{aligned}$$

Letting $h(x) = n - x$, and applying Lemma 6.22, we have $2^{n-x+2^{n-x}} < 2^{n-x+1+2^{n-x+1}}$. So

$$\left(2^{n-x+2^{n-x}} - 2^{n-x+1+2^{n-x+1}}\right) \cdot \ln 4 < 0\,.$$

□

The following theorem proves the existence of the threshold function θ, without giving insight into how to calculate it. See Section 6.1.8 for a discussion on how to calculate it.

Theorem 6.24. *For each $n > 0$, there exists an integer θ, such that*

$$|ROBDD_n| = (2^{n-\theta} - 1) + 2^{2^\theta}\,.$$

Proof. As i increases, so does ${}^n r_i = 2^{i-1}$. By Lemma 6.23, ${}^n R_i = 2^{2^{n-i+1}} - 2^{2^{n-i}}$ is decreasing (as a function of i). At $i = 0$, $2^{i-1} < 2^{2^{n-i+1}} - 2^{2^{n-i}}$. So there necessarily exists a χ_n such that when $i < \chi_n$ we have ${}^n r_i < {}^n R_i$, and when $i \geqslant \chi_n$ we have ${}^n r_i \geqslant {}^n R_i$.

$$\begin{aligned} |ROBDD_n| &= 2 + \sum_{i=1}^{n} \min\{{}^n r_i, {}^n R_i\} && \text{by 6.5} \\ &= 2 + \sum_{i=1}^{\chi_n - 1} {}^n r_i + \sum_{i=\chi_n}^{n} {}^n R_i && (6.7) \end{aligned}$$

Definition 6.25. We define $\theta_n = n - \chi_n + 1$, *i.e.*, the number of terms in the second summation of Equation 6.7.

We also adjust the iteration variable of the second summation to commence at 0. Finally, we apply Lemma 6.16. Simply as a matter of notation, and to facilitate ease of reading, we will write θ rather than θ_n.

$$\begin{aligned} |ROBDD_n| &= 2 + \sum_{i=1}^{n-\theta} {}^n r_i + \sum_{i=n-\theta+1}^{n} {}^n R_i \\ &= 2 + \sum_{i=1}^{n-\theta} {}^n r_i + \sum_{k=0}^{\theta-1} {}^n R_{n-k} \\ &= 2 + \sum_{i=1}^{n-\theta} 2^{i-1} + \sum_{k=0}^{\theta-1} (2^{2^{k+1}} - 2^{2^k}) \end{aligned}$$

Notice that $\sum_{i=1}^{n-\theta} 2^{i-1}$ is a truncated geometric series whose sum is $2^{n-\theta} - 1$. Furthermore, $\sum_{k=0}^{\theta_n - 1} (2^{2^{k+1}} - 2^{2^k})$ is a telescoping series for which all adjacent terms cancel, leaving the difference $2^{2^\theta} - 2^{2^0} = 2^{2^\theta} - 2$. Thus we have the desired equality.

$$\begin{aligned} |ROBDD_n| &= 2 + (2^{n-\theta} - 1) + (2^{2^\theta} - 2) \\ &= (2^{n-\theta} - 1) + 2^{2^\theta} \end{aligned}$$

□

This result makes sense intuitively. The $(2^{n-\theta} - 1)$ term represents the exponential growth of the ROBDD seen in the top rows, from row 1 to row $n - \theta$, as can be seen in the illustrations such as Figure 6.29. The 2^{2^θ} term represents the double-exponential decay in the bottom rows as can be seen in the same illustration.

Definition 6.26. The function $\psi : \mathbb{R}^+ \mapsto \mathbb{R}$ is the function such that

$$2^{2^{\psi(n)+1}} - 2^{2^{\psi(n)}} = 2^{n-\psi(n)-1} . \tag{6.8}$$

Notation 6.27. The largest integer less than x is denoted $\lfloor x \rfloor$.

Notation 6.28. The smallest integer greater than x is denoted $\lceil x \rceil$.

Another way to think of θ is as follows. We define the integer sequence θ_n as the corresponding values of the real valued function ψ defined in Definition 6.26

$$\theta_n = \lfloor \psi(n) \rfloor . \tag{6.9}$$

Equation 6.8 is the real number extension of the integer equation ${}^n r_{\theta_n} = {}^n R_{\theta_n}$. Clearly, θ and ψ are functions of n, hence we denote them as such. We will, as before, dispense with the notation when it is clear, and simply refer to θ_n as θ, and $\psi(n)$ as ψ.

Although we do not attempt to express θ in closed form as a function of n, we do know several things about that function. For example we see in Theorem 6.30 that θ is non-decreasing. We also see in Theorems 6.32 and 6.33 that θ is bounded above and below by functions which themselves go to infinity. Thus, θ becomes arbitrarily large (Equation 6.22).

That $\theta = \lfloor \psi \rfloor$, means that θ is the integer such that $n - \theta$ is the maximum integer for which

$${}^n r_{n-\theta} \leqslant {}^n R_{n-\theta} . \tag{6.10}$$

If $n - \theta$ is the maximum such integer, then

$${}^n r_{n-\theta+1} > {}^n R_{n-\theta+1} . \tag{6.11}$$

As an example, consider the case of $n = 3$.

$$\begin{aligned} {}^3 r_2 &= 2 < {}^3 R_2 = 12 \\ {}^3 r_3 &= 4 > {}^3 R_3 = 2 \end{aligned}$$

We see that ${}^3 r_2$ is the largest value of ${}^3 r_i$ which is less than ${}^3 R_i$. So we have $n - \theta = 3 - \theta = 2$, or $\theta = 1$. If we look at the case of $n = 2$ we see why Inequality 6.10 is not a strict inequality.

$$\begin{aligned} {}^2 r_1 &= 1 < {}^2 R_1 = 12 \\ {}^2 r_2 &= 2 \leqslant {}^2 R_2 = 2 \\ {}^2 r_3 &= 4 > {}^2 R_3 = 2 \end{aligned}$$

The equality case of Equation (6.10) can be seen in Figure 6.29, in which the worst-case ROBDD for $n = 2$ has two nodes for Z_2. There are two nodes for two reasons: because $2^{2-1} = 2$ and also because $2^{\underline{2}} = 2$.

The threshold function is non-decreasing

This section establishes that θ (defined by Equation 6.9) is a non-decreasing sequence. In Section 6.1.8, we will show by Theorems 6.32 and 6.33 that θ is bounded above and below by increasing functions. However, this reasoning alone is not sufficient to show that θ itself is non-decreasing.

To show θ is non-decreasing (Theorem 6.30), we first show that ψ, as Definition 6.26, is strictly increasing (Lemma 6.29). To prove Lemma 6.29, we need two identities, proven earlier in Lemmas 6.20 and 6.21.

Lemma 6.29. *$\psi : \mathbb{R}^+ \mapsto \mathbb{R}$ is strictly increasing.*

Proof. To show that ψ is increasing, we show that its derivative, $\frac{d}{dx}\psi(x)$, is strictly positive. We use x as the variable of differentiation rather than n to emphasize that the domain of ψ is $\mathbb{R}^+$ not $\mathbb{N}$. Note that it is not actually necessary to calculate the derivative of ψ in a form independent of ψ. Rather, it suffices to show that the derivative is positive. We find an expression for $\frac{d}{dx}\psi(x)$ in terms of $\psi(x)$ using implicit differentiation.

$$\begin{aligned} 2^{2^{\psi(x)+1}} - 2^{2^{\psi(x)}} &= 2^{x-\psi(x)-1} \\ \frac{d}{dx}2^{2^{\psi(x)+1}} - \frac{d}{dx}2^{2^{\psi(x)}} &= \frac{d}{dx}2^{x-\psi(x)-1} \end{aligned} \tag{6.12}$$

For clarity, we calculate these three derivatives separately. Applications of Lemma 6.20 and Lemma 6.21 lead to:

$$\frac{d}{dx}2^{2^{\psi+1}} = 2^{\psi+1+2^{\psi+1}} \cdot \ln 4 \cdot \frac{d\psi}{dx} \tag{6.13}$$

$$\frac{d}{dx}2^{2^{\psi}} = 2^{\psi+2^{\psi}} \cdot \ln 4 \cdot \frac{d\psi}{dx} \tag{6.14}$$

$$\frac{d}{dx}2^{x-\psi-1} = 2^{x-\psi-1} \cdot \ln 2 \cdot (1 - \frac{d\psi}{dx}) \tag{6.15}$$

Substituting 6.13, 6.14, and 6.15 into 6.12, and solving for $\frac{d\psi}{dx}$ results in

$$\frac{d\psi}{dx} = \frac{2^{x-\psi-1}}{\ln 2 \cdot (2^{\psi+1+2^{\psi+1}} - 2^{\psi+2^{\psi}}) + 2^{x-\psi-1}} \,. \tag{6.16}$$

Since the right hand side of Equation 6.16 is a fraction whose numerator, $2^{x-\psi-1}$, is positive, and whose denominator is the sum of two terms, the second of which, $2^{x-\psi-1}$, is positive, then it suffices to argue that the first term in the denominator, $\ln 2 \cdot (2^{\psi+1+2^{\psi+1}} - 2^{\psi+2^{\psi}})$, is positive. If we let $h(x) = \psi(x) + 1$, then Lemma 6.22 implies $2^{\psi+1+2^{\psi+1}} > 2^{\psi+2^{\psi}}$. So since $\ln 2 > 0$, we conclude that $\ln 2 \cdot (2^{\psi+1+2^{\psi+1}} - 2^{\psi+2^{\psi}}) > 0$.

$$\frac{d\psi}{dx} = \frac{\overbrace{2^{x-\psi-1}}^{>0}}{\underbrace{\ln 2}_{>0} \cdot \underbrace{(2^{\psi+1+2^{\psi+1}} - 2^{\psi+2^{\psi}})}_{\psi+1+2^{\psi+1} \;>\; \psi+2^{\psi}} + \underbrace{2^{x-\psi-1}}_{>0}} \quad > 0 \,.$$

□

Theorem 6.30. *$\theta : \mathbb{N} \mapsto \mathbb{N}$ by $\theta_n = \lfloor \psi(n) \rfloor$ is non-decreasing.*

Proof. $\psi : \mathbb{R}^+ \mapsto \mathbb{R}$ is increasing (Lemma 6.29), implies that if $m \in \mathbb{N}$, then $\psi(m+1) > \psi(m)$. Thus $\lfloor \psi(m+1) \rfloor \geqslant \lfloor \psi(m) \rfloor$; *i.e.*, $\theta_{m+1} \geqslant \theta_m$ holds for all $m \in \mathbb{N}$. □

Bounds for the threshold function

We showed in Theorem 6.24 that the function θ is well defined, but we didn't say how to calculate it. We now show that θ is bounded by two logarithmic functions. Using those bounds, we will then develop an efficient algorithm for calculating it iteratively (Section 6.1.8). To do this, we first establish an inequality (Lemma 6.31) to be used later.

Lemma 6.31. *For any real number $\alpha > 0$, we have*

$$2^{2^\alpha} < 2^{2^{\alpha+1}} - 2^{2^\alpha}\,.$$

Proof.

$$\begin{aligned}
1 = 2^0 &< 2^\alpha \\
2 = 2^1 &< 2^{2^\alpha} \\
1 &< 2^{2^\alpha} - 1 \\
&= 2^{(2-1)\cdot 2^\alpha} - 1 \\
&= 2^{2\cdot 2^\alpha - 2^\alpha} - 1 \\
&= 2^{2^{\alpha+1} - 2^\alpha} - 1 \\
\frac{2^{2^\alpha}}{2^{2^\alpha}} &< \frac{2^{2^{\alpha+1}}}{2^{2^\alpha}} - \frac{2^{2^\alpha}}{2^{2^\alpha}} \\
2^{2^\alpha} &< 2^{2^{\alpha+1}} - 2^{2^\alpha}
\end{aligned}$$

□

We now establish an upper bound for θ.

Theorem 6.32. *For any $n \in \mathbb{N}$, we have*

$$\theta_n < \log_2 n\,.$$

Proof.

$$\begin{aligned}
{}^nR_{n-\theta+1} &< {}^nr_{n-\theta+1} && \text{by 6.11} \\
2^{2^{\theta+1}} - 2^{2^\theta} &< 2^{n-\theta} \\
2^{2^\theta} < 2^{2^{\theta+1}} - 2^{2^\theta} &< 2^{n-\theta} && \text{by Lemma 6.31} \\
2^\theta &< n - \theta < n \\
\theta &< \log_2 n
\end{aligned}$$

□

We now establish a lower bound for θ.

Theorem 6.33. *For any $n \in \mathbb{N}$, we have*

$$\log_2(n - 2 - \log_2 n) - 2 \leqslant \theta\,.$$

Proof.

$$
\begin{aligned}
\psi - 1 \leqslant \lfloor \psi \rfloor = \theta &< \log_2 n \\
\psi &< 1 + \log_2 n && (6.17)\\
\psi + 1 &< \theta + 2 && (6.18)\\
2^{n-2-\log_2 n} &= 2^{n-(1+\log_2 n)-1} && \text{by 6.4 and Lemma 6.16} && (6.19)\\
&< 2^{n-\psi-1} && \text{by 6.17}\\
&= 2^{2^{\psi+1}} - 2^{2^{\psi}} && \text{by 6.8}\\
&< 2^{2^{\psi+1}}\\
&< 2^{2^{\theta+2}} && \text{by 6.18}\\
2^{\theta+2} &> n - 2 - \log_2 n\\
\theta &> \log_2(n - 2 - \log_2 n) - 2 && (6.20)
\end{aligned}
$$

□

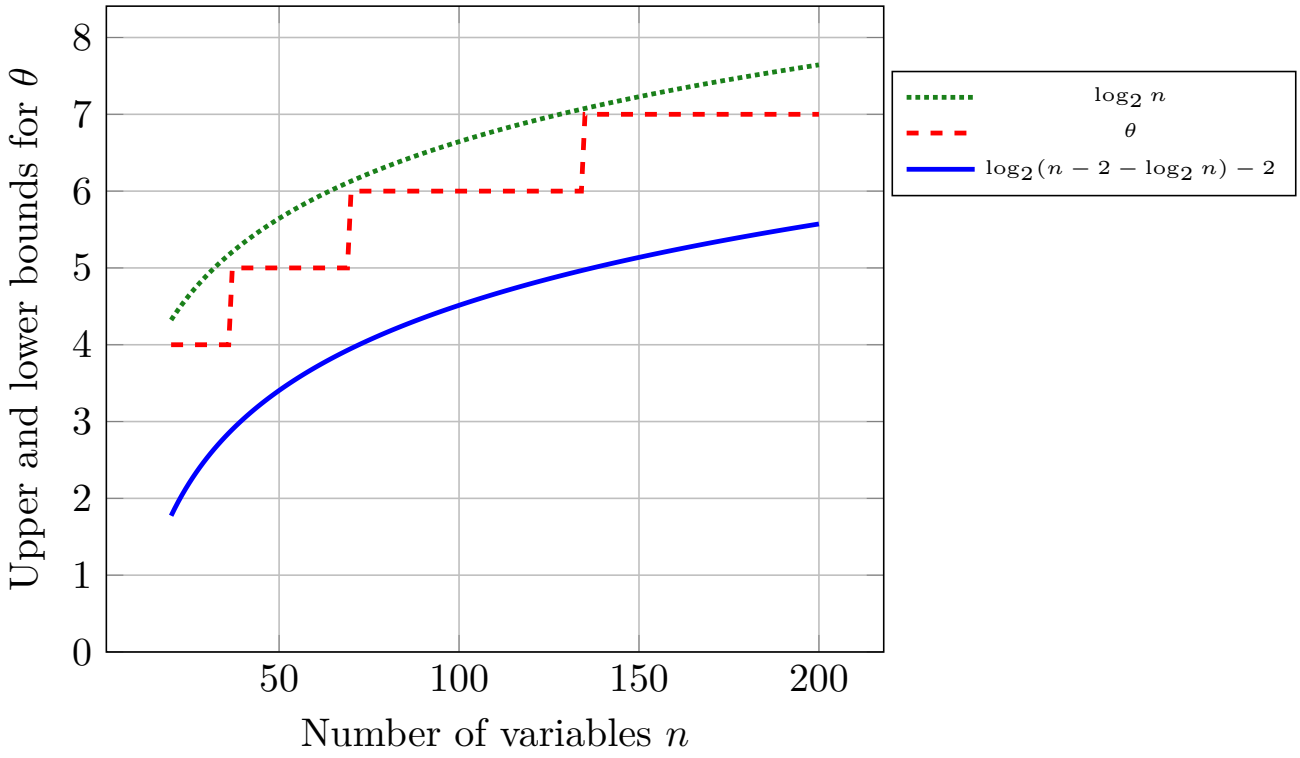


Figure 6.32: Upper and lower bounds for θ

As a consequence of Theorems 6.32 and 6.33, Corollary 6.34 defines upper and lower bounds for θ. The continuous, real valued bounds are illustrated in Figure 6.32.

Corollary 6.34. *For any $n \in \mathbb{N}$, we have*

$$\lceil \log_2(n - 2 - \log_2 n) \rceil - 2 \leqslant \theta \leqslant \lfloor \log_2 n \rfloor$$

Proof. From Theorems 6.32 and 6.33 we already have

$$\log_2(n - 2 - \log_2 n) - 2 \leqslant \theta \leqslant \log_2 n\,,$$

but since θ is an integer, the inequality implies

$$\lceil \log_2(n - 2 - \log_2 n) \rceil - 2 \leqslant \theta \leqslant \lfloor \log_2 n \rfloor$$

□

As is implied by Figure 6.32, and as proven in Theorem 6.35, $\theta \to \infty$.

Theorem 6.35.

$$\lim_{n\to\infty} \theta_n = \infty$$

Proof. First note that for $n \gg 0$

$$\log_2 n < \frac{n}{2}\,. \tag{6.21}$$

Next, we have a lower bound for θ,

$$\begin{aligned}
\theta_n &\geqslant \log_2(n-2-\log_2 n) - 2 && \text{by 6.20}\\
\lim_{n\to\infty} \theta_n &\geqslant \lim_{n\to\infty} \log_2(n-2-\log_2 n) - 2\\
&\geqslant \lim_{n\to\infty} \log_2(n-2-\frac{n}{2}) - 2 && \text{by 6.21}\\
&= \lim_{n\to\infty} \log_2(\frac{n}{2}-2) - 2\\
&= \infty && (6.22)
\end{aligned}$$

□

Corollary 6.36.

$$\lim_{n\to\infty} \psi(n) = \infty\,.$$

Proof. Since

$$\theta_n = \lfloor \psi(n) \rfloor \leqslant \psi(n) \leqslant \lceil \psi(n) \rceil \leqslant 1 + \theta_n\,,$$

then by application of Theorem 6.35 we have

$$\begin{aligned}
\infty = \lim_{n\to\infty} \theta_n &\leqslant \lim_{n\to\infty} \psi(n)\\
&\leqslant \lim_{n\to\infty} 1 + \theta_n = \infty\,.
\end{aligned}$$

So,

$$\lim_{n\to\infty} \psi(n) = \infty\,. \tag{6.23}$$

□

Computing the threshold function

For a given n, the value of θ can be found iteratively, as shown in Algorithm 3. Initializing θ to the upper bound $\lfloor \log_2(n) \rfloor$ as initial guess, from Corollary 6.34, we continue to decrement θ as long as $2^{2^{\theta+1}} - 2^{2^\theta} < 2^{n-\theta-1}$. This iteration usually seems to terminate after 2 iterations. When Algorithm 3 runs from $n = 2$ to $n = 200001$, it terminates after 3 iterations 152 times, and after 2 iterations 199848 times (99.92%). Figure 6.33 shows the values of θ for $1 \leqslant n \leqslant 21$ as calculated by Algorithm 3.

Algorithm 3 terminates in about two iterations, which makes sense when Theorem 6.37 is considered. We see in that theorem that for large n the difference of the upper and lower limits expressed in Corollary 6.34 is 2. However, we see from experimentation that a small fraction of the time the algorithm terminates at 3 iterations. This is because Algorithm 3 arranges that θ is always decremented once too many (except when $n = 1$). This is why $\theta + 1$ is returned on line 3.10 of Algorithm 3.

Theorem 6.37. *For all sufficiently large n,*

$$\log_2 n - \theta_n < 2\,.$$

n	$\lfloor \log_2 n \rfloor$	θ	$2^{n-\theta} - 1 + 2^{2^\theta}$	n	$\lfloor \log_2 n \rfloor$	θ	$2^{n-\theta} - 1 + 2^{2^\theta}$
1	0	0	3	20	4	4	131,071
2	1	1	5	30	4	4	67,174,399
3	1	1	7	50	5	5	3.52×10^{31}
4	2	1	11	100	6	6	1.98×10^{28}
5	2	1	19	200	7	7	1.26×10^{58}
6	2	2	31	500	8	8	1.28×10^{148}
7	2	2	47	1000	9	9	2.09×10^{298}
8	3	2	79	2000	10	10	1.12×10^{599}
9	3	2	143	5000	12	12	3.45×10^{1501}
10	3	2	271	10,000	13	13	2.43×10^{3006}
11	3	3	511	20,000	14	14	2.42×10^{6016}

Figure 6.33: Worst-case ROBDD size, $|ROBDD_n|$, in terms of number of variables, n. The table also shows θ (the threshold) and $\lfloor \log_2 n \rfloor$ demonstrating that $\lfloor \log_2 n \rfloor$ serves both as an upper bound and as an initial guess for θ. The table also shows the exponential term and the double-exponential term, whose sum is the worst-case size.

Proof. We know that for $n \gg 0$, θ_n lies between the upper and lower bounds indicated in Theorems 6.32 and 6.33. This means

$$\log_2 n - \theta_n < \log_2 n - (\log_2(n - 2 - \log_2 n) - 2)\,.$$

$$\begin{aligned}
\Delta_{bounds} &= \lim_{n\to\infty} \Big(\overbrace{\log_2 n}^{\text{upper bound}} - \underbrace{\big(\log_2(n-2-\log_2 n) - 2\big)}_{\text{lower bound}} \Big) \\
&= 2 + \lim_{n\to\infty} \log_2 \frac{n}{n-2-\log_2 n} \\
&= 2 + \log_2 \lim_{n\to\infty} \frac{n}{n-2-\log_2 n} \quad = \quad 2 + \log_2 \lim_{n\to\infty} \frac{\overbrace{\frac{d}{dn} n}^{\text{L'Hôpital's rule}}}{\frac{d}{dn}(n-2-\log_2 n)} \\
&= 2 + \log_2 \lim_{n\to\infty} \frac{1}{1-\frac{1}{n}} = 2 + \log_2 1 = 2
\end{aligned}$$

□

Algorithm 3: FINDTHETA determine θ iteratively

Input: n: positive integer $n > 0$, indicating the number of Boolean variables
Output: θ: minimum integer, θ such that ${}^n r_{n-\theta} \leqslant {}^n R_{n-\theta}$; *i.e.*, $2^{n-\theta} \leqslant 2^{2^\theta} - 2^{2^{\theta-1}}$

```
3.1  begin
3.2      if n = 1 then
3.3          return 0
3.4      θ ← ⌊log2 n⌋ + 1
3.5      repeat
3.6          θ ← θ − 1
3.7          r ← 2^(n−θ−1)
3.8          R ← 2^(2^(θ+1)) − 2^(2^θ)
3.9      until R < r
3.10     return θ + 1
```

6.1.9 Plots of $|ROBDD_n|$ and related quantities

Now that we can calculate θ, it is possible to plot $|ROBDD_n|$ as a function of n. The plots in Figure 6.34 show the relative sizes of $2^{n-\theta}$, 2^{2^θ}, and their sum $|ROBDD_n|$. The plot also shows 2^n, which is intended to convey intuition about relative sizes of the various quantities. In the plot on the right, it appears that $2^{n-\theta}$ becomes a good approximation for $|ROBDD_n|$ for large values of n. However, the plot on the left shows that this is a poor approximation for values of n below 15.

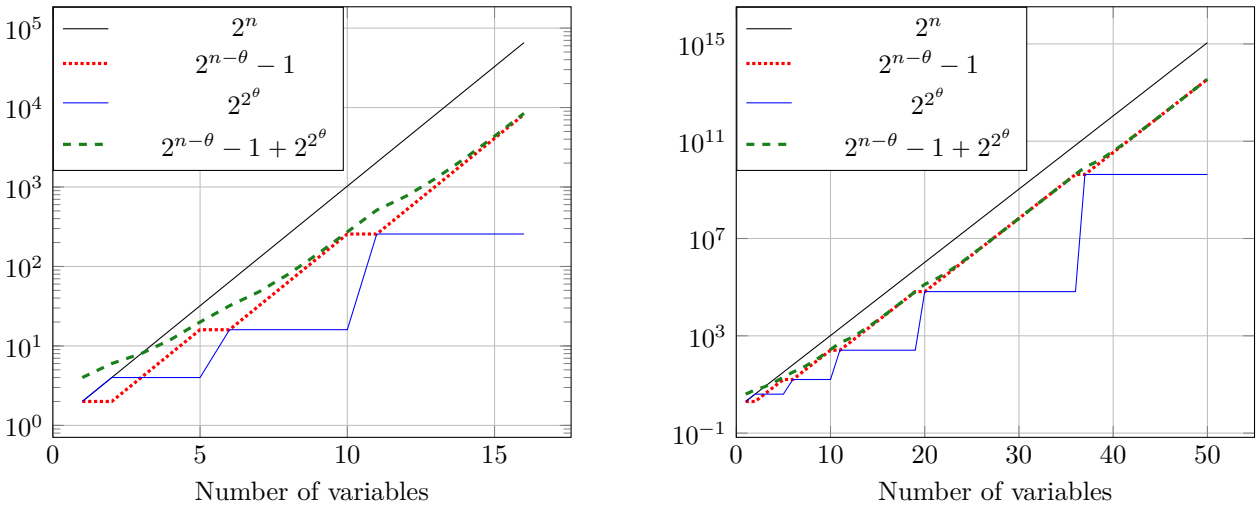


Figure 6.34: The plots show the relative sizes of $2^{n-\theta}$, 2^{2^θ}, and their sum $|ROBDD_n|$.

6.1.10 Limit of the residual compression ratio

In Section 6.1.2, we introduced ρ_n, the ROBDD residual compression ratio (Equation 6.1). We also observed in Figure 6.28 that ρ_n seems to decrease as n increases. Moreover, Figure 6.35 shows ρ_n calculated by Equation 6.1 for values of $1 \leqslant n \leqslant 21$. The plot in Figure 6.35 shows the residual compression ratio for $1 \leqslant n \leqslant 200$. In this plot, it appears that the residual compression ratio tends to 0. This is in fact the case, as proven in Theorem 6.38.

n	$\lvert ROBDD_n\rvert$	ρ_n
1	3	100.000%
2	5	71.429%
3	7	46.667%
4	11	35.484%
5	19	30.159%
6	31	24.409%
7	47	18.431%
8	79	15.460%
9	143	13.978%
10	271	13.239%
11	511	12.479%
12	767	9.364%
13	1279	7.807%
14	2303	7.028%
15	4351	6.639%
16	8447	6.445%
17	16639	6.347%
18	33023	6.299%
19	65791	6.274%
20	131071	6.250%
21	196607	4.687%

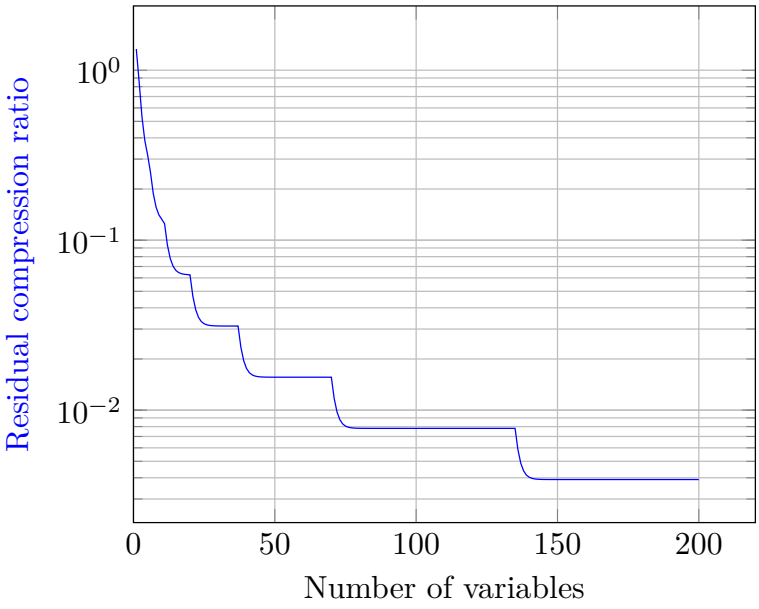


Figure 6.35: Residual compression ratio of worst-case ROBDD, calculated from theoretical data as compared to UOBDD, and shown in tabular graphical form.

Theorem 6.38.

$$\lim_{n\to\infty} \rho_n = 0\,.$$

Proof. First, we establish a few helpful inequalities.

$$\begin{aligned} |UOBDD_n| &= 2^{n+1}-1 && \text{by 5.1} \\ &> 2^n && \text{for } n \gg 0 \end{aligned} \tag{6.24}$$

$$
\begin{aligned}
2^{2^{\theta}} = 2^{2^{\lfloor \psi(n) \rfloor}} &\leqslant 2^{2^{\psi(n)}} && \text{by 6.9} \\
&< 2^{2^{\psi(n)+1}} - 2^{2^{\psi(n)}} && \text{by Lemma 6.31} && (6.25) \\
&= 2^{n-\psi(n)-1} && \text{by 6.8} && (6.26)
\end{aligned}
$$

$$
\begin{aligned}
|ROBDD_n| &= 2^{2^{\theta_n}} - 2^{n-\theta_n} - 1 && \text{by Theorem 6.24} \\
&\leqslant 2^{n-\psi(n)-1} - 2^{n-\theta_n} - 1 && \text{by 6.26} && (6.27)
\end{aligned}
$$

$$
\begin{aligned}
\rho_n &= \frac{|ROBDD_n|}{|UOBDD_n|} && \text{by 6.1} \\
&< \frac{|ROBDD_n|}{2^n} && \text{by 6.24} \\
&\leqslant \frac{2^{n-\psi(n)-1} - 2^{n-\theta_n} - 1}{2^n} && \text{by 6.27} && (6.28)
\end{aligned}
$$

Now, we can apply the limit to Inequality 6.28.

$$
\begin{aligned}
\lim_{n\to\infty} \rho_n &\leqslant \lim_{n\to\infty} \frac{2^{n-\psi(n)-1} - 2^{n-\theta_n} - 1}{2^n} \\
&= \lim_{n\to\infty} \frac{2^{n-\psi(n)-1}}{2^n} - \lim_{n\to\infty} \frac{2^{n-\theta_n}}{2^n} - \lim_{n\to\infty} \frac{1}{2^n} \\
&= \lim_{n\to\infty} \frac{1}{2^{\psi(n)+1}} - \lim_{n\to\infty} \frac{1}{2^{\theta_n}} - \lim_{n\to\infty} \frac{1}{2^n} \\
&\leqslant 0 - 0 - 0 && \text{by 6.23 and 6.22} \\
\lim_{n\to\infty} \rho_n &\leqslant 0\,.
\end{aligned}
$$

Since for each n, ρ_n is the quotient of two positive numbers, we know that $\rho_n > 0$. We can thus conclude that

$$\lim_{n\to\infty} \rho_n = 0\,.$$

□

6.2 Programmatic construction of a worst-case n-variable ROBDD

In the previous sections, we looked at various examples of ROBDDs of different sizes. During our experimentation, we found it necessary to devise an algorithm for generating worst-case ROBDDs. Because worst-case ROBDDs are not unique, any such algorithm has leeway in the manner it constructs them. In this section, we discuss the algorithm we developed, *i.e.*, an algorithm for constructing a worst-case ROBDD of n Boolean variables, denoted $Z_1, Z_2, ...Z_n$. The constructed ROBDDs resemble those shown in Figure 6.29.

Recall, from Section 6.1.7, that the worst-case ROBDD can be thought of as having two parts, which we will call the top part and the bottom part. The top and bottom parts are illustrated in Figure 6.30. The top part comprises the exponential expansion from the root node (corresponding to Z_1) called row 0, and continuing until row $\mathcal{B}$. Recall the definition of $\mathcal{B}$ in Definition 6.9.

$$\mathcal{B} = n - \theta - 1\,. \tag{6.29}$$

This top part contains $n - \theta$ number of rows. The bottom part comprises the double-exponential (*i.e.*, 2^{2^i}) decay, starting at row $n - \theta$, and continuing through row n for a total of $\theta + 1$ rows. The bottommost row is row n which contains precisely the two singleton objects $\top$ and $\bot$. From this information and the following few notational definitions, we are able to construct one of the many possible worst-case n-variable ROBDDs with Algorithm 4.

Notation 6.39. An ROBDD node is denoted as $node(\top)$ (true terminal node), $node(\bot)$ (false terminal node) or $node(i, \alpha, \beta)$ (non-terminal node on row_i with child nodes α and β.)

Notation 6.40. Let row_a^b denote the set of nodes in any of rows a to b.

Notation 6.41. If S is a set, with cardinality $|S|$, let

$$\mathscr{P}(S) = \{(\alpha, \beta) \mid \alpha, \beta \in S, \alpha \neq \beta\}.$$

$\mathscr{P}(S)$ is the set of non-duplicate pairs chosen from S.

Notice that $|\mathscr{P}(S)| = |S|^2$.

Algorithm 4 generates an ROBDD represented as a vector of sets of nodes $[row_0, row_1, ..., row_n]$. Lines 4.4 through 4.6 generate the bottom part, and lines 4.8 through 4.11 generate the top part. Algorithm 5 generates the belt as illustrated in Figure 6.30.

Algorithm 4: GenWorstCaseROBDD generates a worst-case ROBDD

Input: n, positive integer indicating the number of Boolean variables
Output: a vector of sets of nodes

4.1 $\theta \leftarrow FindTheta(n)$ // Algorithm 3
4.2 $\mathcal{B} \leftarrow n - \theta - 1$ // belt row index
4.3 // Generate the bottom part
4.4 $row_n \leftarrow \{node(\top), node(\bot)\}$ // row of leaves
4.5 **for** i from $n-1$ downto $\mathcal{B}+1$ **do**
4.6 $row_i \leftarrow \{node(i, \alpha, \beta) \mid (\alpha, \beta) \in \mathscr{P}(row_{i+1}^n)\}$ // $|row_i| = {}^nR_i$
4.7 // Generate the top part
4.8 $row_{\mathcal{B}} \leftarrow GenBelt(n, \mathcal{B}, row)$ // Algorithm 5
4.9 **for** i from $\mathcal{B}-1$ downto 0, **do**
4.10 $P \leftarrow$ any partition of row_{i+1} into ordered pairs // possible by Theorem 6.18, $|P| = \frac{|row_{i+1}|}{2} = 2^i$
4.11 $row_i \leftarrow \{node(i, \alpha, \beta) \mid (\alpha, \beta) \in P\}$ // $|row_i| = 2^i$
4.12 **return** $[row_0, row_1, ..., row_n]$ // generated $1 + (n - \mathcal{B} - 1) + 1 + \mathcal{B} = n + 1$ rows.

Algorithm 5: GenBelt generates the $\mathcal{B}$ row of the worst-case ROBDD

Input: n, positive integer indicating the number of Boolean variables
Input: $\mathcal{B}$, between 0 and n, indicating the belt's row number
Input: row, a vector which the calling function has partially populated. $row_{\mathcal{B}+1}...row_n$ are each non-empty sets of nodes.
Output: a set of $2^{\mathcal{B}}$ nodes intended to comprise $row_{\mathcal{B}}$

5.1 $p \leftarrow |row_{\mathcal{B}+1}|$ // calculate ${}^nR_{\mathcal{B}+1}$
5.2 $P_{left} \leftarrow$ any partition of $row_{\mathcal{B}+1}$ into ordered pairs // possible by Theorem 6.18
5.3 $S_{left} \leftarrow \{node(\mathcal{B}, \alpha, \beta) \mid (\alpha, \beta) \in P_{left}\}$
5.4 **if** $2^{\mathcal{B}} < p \cdot (p-1)$ **then** // if wide belt
5.5 $P_{right} \leftarrow \mathscr{P}(row_{\beta+1}^n)$
5.6 **else** // if narrow belt
5.7 $P_{right} \leftarrow \mathscr{P}(row_{\beta+1})$
// We want $|row_{\mathcal{B}}| = 2^{\mathcal{B}}$. So limit $|S_{right}|$ to $2^{\mathcal{B}} - \frac{{}^nR_{\mathcal{B}+1}}{2}$.
5.8 $S_{right} \leftarrow$ any $\left(2^{\mathcal{B}} - |S_{left}|\right)$ sized subset of $\{node(\mathcal{B}, \alpha, \beta) \mid (\alpha, \beta) \in P_{right} \backslash P_{left}\}$
5.9 **return** $S_{left} \cup S_{right}$ // $|S_{left}| + |S_{right}| = \frac{{}^nR_{\mathcal{B}+1}}{2} + (2^{\mathcal{B}} - \frac{{}^nR_{\mathcal{B}+1}}{2}) = 2^{\mathcal{B}}$

For simplicity, we don't specify how to perform the computations on lines 4.10, 5.2, and 5.8. Solutions may vary, depending on the choice of programming language and data structures. Lines 4.10 and 5.2 call for a set

with an even number of elements to be partitioned into pairs. Such a partitioning can be done in many different ways, one of those being

$$\{node_1, node_2 ... node_m\} \mapsto \{(node_1, node_2), (node_3, node_4) ... (node_{m-1}, node_m)\}$$

Line 5.8 calls for the generation of any subset of a given size, and given a specified superset. In particular it asks for a subset,

$$S_{right} \subseteq \{node(\mathcal{B}, \alpha, \beta) \mid (\alpha, \beta) \in P_{right} \backslash P_{left}\}\,, \text{ such that } |S_{right}| = 2^{\mathcal{B}} - |S_{left}|\,.$$

One way to generate such a subset might be to first generate the superset, then truncate it to the desired size. A more clever way would be to start as if generating the superset, but stop once a sufficient number of elements is reached.

In Algorithm 5, there are two cases to consider. The question posed on line 5.4 is whether it is possible to generate $p^2 = p \cdot (p-1)$ unique ordered pairs of nodes from $row_{\mathcal{B}+1}$, possibly with some left over.

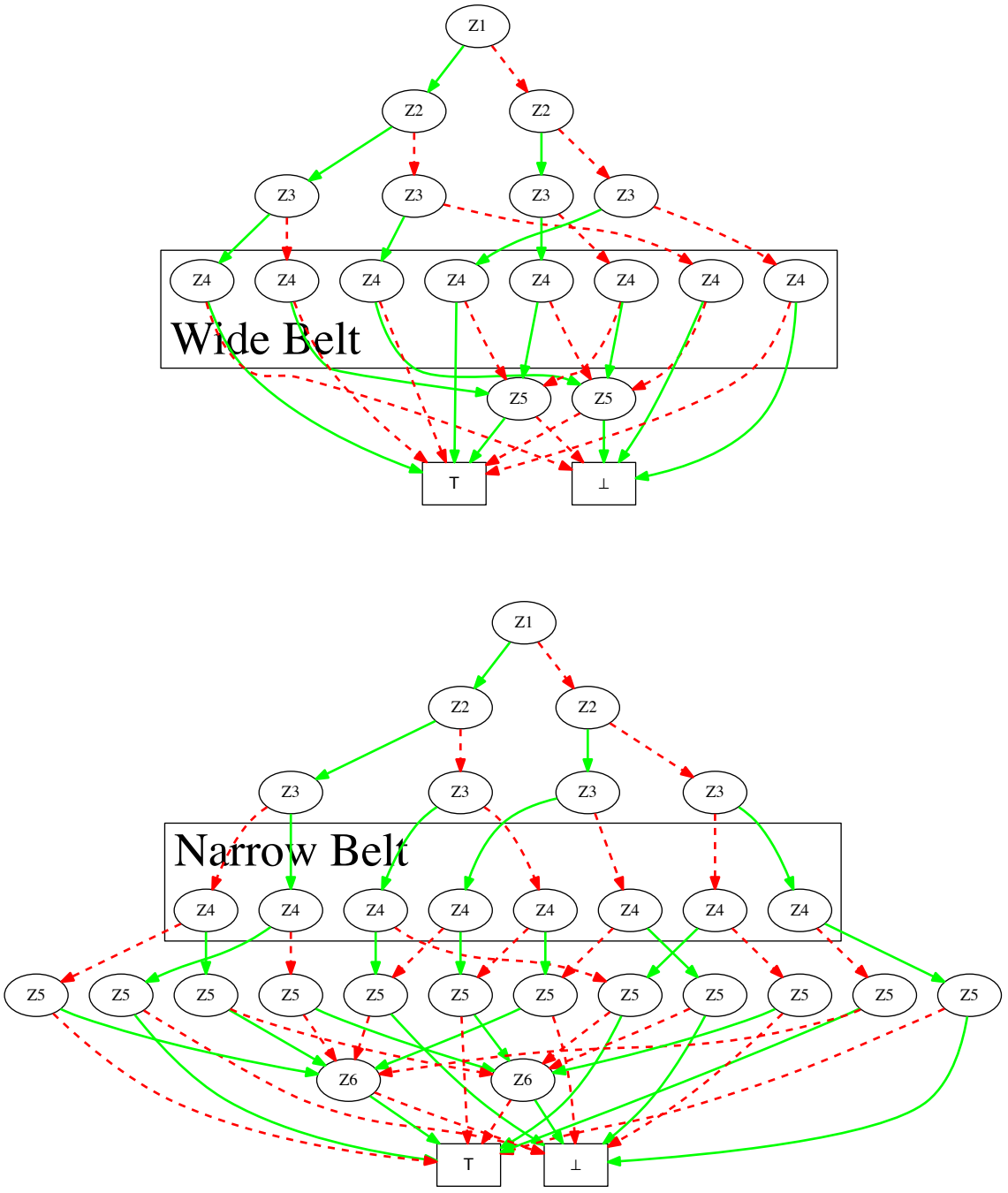


Figure 6.36: Belt connections of the ROBDD for 5 and 6 variables. The narrow belt only connects to the row directly below it. The wide belt connects not only to the row below it, but also to other, lower rows.

Wide: $2^{\mathcal{B}} > p \cdot (p-1)$ The belt is called wide because it touches not only the row directly below it, but others as well.

The Wide Belt case is illustrated in Figure 6.36. In this case $row_{\mathcal{B}}$ is row_3 (corresponding to Z_4) which has $2^3 = 8$ nodes. However, $row_{\mathcal{B}+1}$, *i.e.* row_4, (corresponding to Z_5) has only two nodes. There is an insufficient number of nodes in row_4 to connect the nodes in row_3. For this reason, connections are made, not only to $row_{\mathcal{B}+1}$, but also to some or all the nodes below it. Line 5.5 collects the set of all ordered pairs of nodes coming from $row_{\mathcal{B}+1}$ to row_n, and we will later (on line 5.8) want to subtract out those ordered pairs already collected in P_{left} to avoid generating congruent nodes. This set of ordered pairs might be large, so our suggestion is to generate a lazy set. Explanations of lazy data structures are numerous (Okasaki [Oka98] and Slade [Sla98, Section 14.6], to name a few).

Narrow: $2^{\mathcal{B}} \leqslant p \cdot (p-1)$ The belt is called narrow because unlike the wide belt, it only touches the row directly below it.

The Narrow Belt case is illustrated in Figure 6.36. In the figure we see that row_3, corresponding to variable Z_4, has 8 nodes, and $2^4 = 16$ arrows pointing downward. Since row_4 does not contain more than 16 nodes, it is possible to connect the belt to the bottom part simply by constructing the connecting arrows exclusively between row_3 and row_4 (between Z_4 and Z_5).

The time complexity of Algorithm 4 may vary, depending on the types of data structures used, and also according to which choices the programmers makes in implementing the set relative complement operation and truncated subset operations on line 5.8. However, in every case, the algorithm must generate $|ROBDD_n|$ number of nodes. The complexity, therefore, cannot be made better than $\Omega(2^{n-\theta} + 2^{2^\theta})$, (we refer the reader to Wegener [Weg87, Section 1.5] for a discussion of Ω notation). The plots in Figure 6.34 convey an intuition of the relative sizes of $2^{n-\theta}$ vs 2^{2^θ}; *i.e.* that for large n, $2^{n-\theta} - 1$ becomes a good approximation for $|ROBDD_n|$. Thus we may approximate $\Omega(2^{n-\theta} + 2^{2^\theta}) \approx \Omega(2^{n-\theta})$.

6.3 Related work

Newton *et al.* [NV18a] presented a preliminary discussion of these results but primarily limited to 10 Boolean variables.

Newton *et al.* [NVC17] discuss calculations related to the Common Lisp type system. These calculations are facilitated using ROBDDs, and there is an implication that Boolean expressions (describing Common Lisp types in this case) are efficient with this choice of data structure. However, no quantification is made in that work to justify the claim. In the current work we treat some of the questions relating to space efficiency using this choice of data structure.

Butler *et al.* [SIHB97] discuss average and worst-case sizes of BDDs representing multiple-valued functions. Miller *et al.* [MD03] also discusses the maximum size of discrete multiple-valued functions.

Bergman and Cire [BC16] discuss size bounds on data structures they call BDDs but which are defined slightly differently than we do. Bergman's data structures only support arcs connecting successive levels (which we call rows). That is, whereas we allow nodes in row i to connect to nodes in any row below it, Bergman only allows connections between rows i and $i+1$. Thus, in Bergman's case, BDDs such as the 4, 5, 6, and 7 variable cases we illustrate in Figure 6.29 are not considered. Nevertheless, we do find that our approach is similar to that of Bergman and Cire in that they both estimate the worst-case width of a row as the minimum of an exponential and a double-exponential, and then proceed to find the threshold where the exponential and double-exponential are equal.

The equation in Theorem 6.24 is similar to that provided by Knuth [Knu09, Section 7.1.4 Theorem U], and that provided by Heap *et al.* [HM94]. The derivation we show here relies heavily on intuitions gleaned from the shapes of worst-case ROBDDs, while the treatment of Knuth relies on a concept called beads which we do not address. The treatment by Heap is indeed similar to our own, albeit less grounded in geometric intuition. Heap's Theorem 1 gives a formula for $R(n)$ which differs by a constant of 2 from our Theorem 6.24. That difference is due to the fact that Heap does not include the two leaf nodes in his calculation as we do. Heap argues that the threshold (his k, our θ) is precisely $\lfloor \log_2 n \rfloor$ or $\lfloor \log_2 n \rfloor - 1$, which seems in keeping with our experimental findings from Algorithm 3 and Figure 6.33.

Gröpl *et al.* [GPS01] improved on the work of Heap by explaining certain oscillations shown but not explained in Heap's work. We also see some variations, which we don't attempt to explain, in size behavior such as in Figure 6.9. These variations may be related to Gröpl's oscillations. They are not apparent until we examine ROBDDs of size more than 10 variables. Additional work by Gröpl *et al.* [GPS98] look again into size of

ROBDDs and discusses the Shannon effect, which explains the correlation between worst-case and average ROBDD sizes.

Minato [Min93] suggests using a different set of reduction rules than those discussed in Section 5.1. The resulting graph is referred to as a Zero-Suppressed BDD, or 0-Sup-BDDs (also referred to as ZDDs and ZBDDs in the literature). Minato claims that this data structure offers certain advantages over ROBDDs in modeling sets and expressing set-related operations, especially in sparse Boolean equations where the number of potential variables is large but the number of variables actually used in most equations is small. Additionally, Minato claims that 0-Sup-BDDs provide advantages when the number of input variables is unknown, which is the case we encounter when dealing with Common Lisp types, because we do not have a way of finding all user defined types.

Lingberg *et al.* [LPR03] consider sizes of ROBDDs representing the very special case of simple CNF (conjunctive normal form) formulas, in particular the representation of CNF formulas consisting of max-terms, each of which consists of exactly two variables, neither of which is negated. He does the majority of his development in terms of QOBDDs and relates back to ROBDDs with his claim that $|ROBDD_n|$ lies between the max size of a QOBDD and half that quantity.

Our work discusses the exponential worst-case ROBDD size of arbitrary Boolean functions of n variables. One might ask whether certain subsets of this space of functions have better worst-case behavior in terms of worst-case ROBDD size. Abío *et al.* [ANO+12] examine Pseudo-Boolean constraints which are integer functions of the form $\sum_{i=1}^{n} a_i x_i \leqslant a_0$ where a_i is an integer and x_i is a Boolean variable. The solution space of such inequalities may be represented by an ROBDD. Abío identifies certain families of Pseudo-Boolean constraints for which the ROBDDs have polynomial size and others which have exponential size.

In our research we consider ROBDD sizes for a fixed variable order. Lozhkin *et al.* [LS10] extends the work of Shannon [Sha49] in examining sizes when allowed to seek a *better* variable ordering.

In Section 6.1.5 we introduced the residual compression ratio. Knuth [Knu09, Section 7.1.4] discusses similar ratios of sizes of BDDs vs ZDDs. Bryant [Bry18] introduces the operation of chain reduction, and discusses size ratios of BDDs and ZDDs to their chain reduced counterparts.

Castagna [Cas16] mentions the use of a lazy union strategy for representing type expressions as BDDs. Here, we have only implemented the strategy described by Andersen [And99]. The Andersen approach involves allocating a hash table to memoize all the BDDs encountered in order both to reduce the incremental allocation burden when new Boolean expressions are encountered, and also to allow occasional pointer comparisons rather than structure comparisons. Castagna suggests that the lazy approach can greatly reduce memory allocation. Additionally, from the description given by Castagna, the lazy union approach implies that some unions involved in certain BDD-related Boolean operations can be delayed until the results are needed, at which time the result can be calculated and stored in the BDD data structure.

Brace *et al.* [BRB90] demonstrate an efficient implementation of a BDD library, complete with details about how to efficiently manage garbage collection (GC). We have not yet seen GC as an issue as our language of choice has a good built-in GC engine which we implicitly take advantage of.

The CUDD [Som] developers put a lot of effort in optimizing their algorithms. Our BDD algorithm can certainly be made more efficient, notably by using techniques from CUDD. The CUDD user manual mentions several interesting and inspiring features. More details are given in Section 6.5.

The sequence $a_n = 2^{2^{n-1}} - 2^{2^{n-2}}$ with $a_1 = 1$ appears in a seemingly unrelated work of Kotsireas and Karamanos [KK04] and shares remarkable similarity to Lemma 6.16. The results of Kotsireas and Karamanos are accessible on the On-Line Encyclopedia of Integer Sequences (OEIS).[1] Using the OEIS we found that the sequence ${}^nR_n, {}^nR_{n-1}, {}^nR_{n-2}, \ldots$ agrees with the Kotsireas sequence from a_2 up to at least a_9, which is a 78 digit integer. This similarity inspired us to investigate whether it was in fact the same sequence, and lead us to pursue the formal development we provide in Section 6.1.7.

6.4 Conclusion

We have provided an analysis of the explicit space requirements of ROBDDs. This analysis includes exhaustive characterization of the sizes of ROBDDs of up to 4 Boolean variables, and an experimental random-sampling approach to provide an intuition of size requirements for ROBDDs of more variables. We have additionally provided a rigorous prediction for the worst-case size of ROBDDs of n variables. We used this size to predict the residual compression the ROBDD provides. While the size itself grows unbounded as a function of n, the residual compression ratio shrinks asymptotically to zero. That is, ROBDDs become arbitrarily more efficient for a sufficiently large number of Boolean variables.

In order to perform our experiments, we had to design an algorithm for generating a worst-case ROBDD for a given number of variables. We have described this algorithm here as well, as having a typical worst-case ROBDD may prove to be useful for other applications than size predictions.

[1] The On-Line Encyclopedia of Integer Sequences or OEIS is available at `https://oeis.org`.

Our approach for this development is different from what we have found in current literature, in that while it is mathematically rigorous, its development is highly based on intuitions gained from experiment.

6.5 Perspectives

There are several obvious shortcomings to our intuitive evaluation of statistical variations in ROBDD sizes as discussed in Section 6.1.2. For example, we stated that judging from the small sample in Figure 6.8, it would appear that for large values of n, $|ROBDD_n|$ is a good estimate for average size. We would like to continue this investigation to better justify this gross approximation.

When using ROBDDs, or presumably 0-Sup-BDDs, one must use a hash table of all the BDDs encountered so far (or at least within a particular dynamic extent). This hash table, mentioned in Section 5.1, is used to assure structural identity. However, it can become extremely large, even if its lifetime is short. Section 6.1 discusses the characterization of the worst-case size of an ROBDD as a function of the number of Boolean variables. This characterization ignores the transient size of the hash table, so one might argue that the size estimations in 6.1 are misleading in practice. We would like to continue our experimentation and analysis to provide ways of measuring or estimating the hash table size, and potentially ways of decreasing the burden incurred. For example, we suspect that most of the hash table entries are never re-used. We would like to experiment with weak hash tables: once all internal and external references to a particular hash table entry have been abandoned, that hash table entry can be removed, thus potentially freeing up the child nodes as well.

As discussed in Section 6.3, Minato [Min93] claims that using the BDD variant called 0-Sup-BDD is well suited for sparse Boolean equations. We see potential applications for this data structure in type calculations, especially when types are viewed as sets, as in Common Lisp. In such cases, the number of types is large, but each type constraint equation scantly concerns few types. We would like to experiment with 0-Sup-BDD based implementations of our algorithms, and contrast the performance results with those found thus far.

It is known that algorithms using BDDs tend to trade space for speed. A question naturally arises: can we implement a fully functional BDD which never stores calculated values. The memory footprint of such an implementation would potentially be smaller, while incremental operations would be slower. It is not clear whether the overall performance would be better or worse. Castagna [Cas16] suggests a lazy version of the BDD data structure which may reduce the memory footprint, which would have a positive effect on the BDD based algorithms. This approach suggests dispensing with the excessive heap allocation necessary to implement Andersen's approach [And99]. Moreover, our implementation (based on the Andersen model) contains additional debug features which increase the memory footprint. We would like to investigate which of these two approaches gives better performance, or allows us to solve certain problems. It seems desirable to attain heuristics to describe situations which one or the other optimization approach is preferable.

Even though both Andersen [And99] and Minato [Min93] claim the necessity to enforce structural identity, it is not clear whether in our case, the run time cost associated with this memory burden, outweighs the advantage gained by structural identity. Furthermore, the approach used by Castagna [Cas16] seems to favor laziness over caching, lending credence to our suspicion.

CUDD [Som] uses a common base data structure, DdNode, to implement several different flavors of BDD, including Algebraic Decision Diagrams (ADDs) and ZDDs. We have already acknowledged the need to experiment with other BDD flavors to efficiently represent run-time type based decisions such as the Common Lisp run-time type reflection [NVC17, NV18c] in performing simplification of type-related logic at compile-time. We wish to examine the question of whether the Common Lisp run-time type reflection can be improved by searching for better ordering of the type specifiers at compile-time. The work of Lozhkin [LS10] and Shannon [Sha49] may give insight into how much improvement is possible, and hence whether it is worth dedicating compilation time to it.

In Section 6.1.4 we examine the question of determining how large a sample size is sufficient. The Engineering Statistics Handbook [Nat10] presents the work of Chakravart *et al.* [Kar68, pp 392-394] who in turn explain the Kolmogorov-Smirnov goodness of fit test. The test is designed to determined whether a sample in question comes from a specific distribution. We would like to apply this test to the sequence of samples in Figures 6.18 through 6.22 to assign a quantitative confidence to the histograms. This is a matter for further research.

Chapter 7

Extending BDDs to Accommodate Common Lisp Types

In Chapter 5 we introduced a data structure called the ROBDD which is useful in tackling many algorithmic problems related to Boolean algebra. Computation dealing with the Common Lisp type system extensively involves Boolean algebra; in particular, computations such as those involved in efficiently recognizing regularly typed sequences introduced in Chapter 4. In the current chapter, we take a necessary step in facilitating those computations. We present an extension of the ROBDD data structure which accommodates Common Lisp types, thus enabling the computations needed in type simplification, type equivalence, and type vacuity checks, among others. In Chapters 9 and 11 we look at solutions related to the MDTD problem and serialization problems introduced in Chapter 4, and the ROBDD will be an important tool in that analysis.

Common Lisp types are most commonly represented in Common Lisp programs as s-expression based type specifiers. In this chapter, we present an alternative internal representation: the Binary Decision Diagram (BDD) [Bry86, Ake78]. BDDs have interesting characteristics such as representational equality; *i.e.* it can be arranged so that equivalent expressions or equivalent sub-expressions are represented by the same object (`eq`). While techniques to implement BDDs with these properties are well documented, an attempt to apply the techniques directly to the Common Lisp type system encounters obstacles which we analyze and document in this chapter.

We encounter a challenge when using s-expressions based algorithms to manipulate type specifiers. It often occurs that after a programmatic manipulation, we need to reduce complex type specifiers to a canonical form. This reduction can be computationally intense, and difficult to implement correctly. The presentation of BDDs in this chapter, obviates much of the need to reduce to canonical form, because the BDD maintains a canonical form by design. Before looking at how the BDD can be used to represent Common Lisp type specifiers, we first look at how BDDs are used traditionally to represent Boolean equations. Thereafter, we explain how this traditional treatment can be enhanced to represent Common Lisp types.

7.1 Representing Boolean expressions

Andersen [And99] summarized many of the algorithms for efficiently manipulating BDDs. Not least important in Andersen's discussion is how to use a hash table and dedicated constructor function to eliminate redundancy within a single BDD and within an interrelated set of BDDs. The result of Andersen's approach is that if you attempt to construct two BDDs to represent two semantically equivalent but syntactically different Boolean expressions, then the two resulting BDDs are pointers to the same object.

Figure 5.1 shows an example BDD illustrating a function of three Boolean variables: A_1, A_2, and A_3. To reconstruct the disjunctive normal form (DNF), collect the paths from the root node, A_1, to a leaf node of 1, ignoring paths terminated by 0. When the right child is traversed, the Boolean complement ($\neg$) of the label on the node is collected (*e.g.* $\neg A_3$), and when the left child is traversed the non-inverted parent is collected. Interpret each path as a conjunctive clause, and form a disjunction of the conjunctive clauses. In the figure the three paths from A_1 to 1 identify the three conjunctive clauses $(A_1 \wedge A_2)$, $(A_1 \wedge \neg A_2 \wedge A_3)$, and $(\neg A_1 \wedge \neg A_3)$.

7.2 Representing types

Castagna [Cas16] explains the connection of BDDs to type theoretical calculations, and provides straightforward algorithms for implementing set operations (intersection, union, relative complement) of types using BDDs. The general algorithms for these operations are presented in Section 5.2

7.3 Representing Common Lisp types

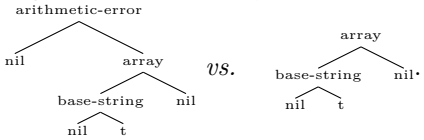
We have implemented the BDD data structure as a set of CLOS classes. In particular, there is one leaf-level CLOS class for an internal tree node, and one singleton class/instance for each of the two possible leaf nodes, *true* and *false*.

The label of the BDD contains a Common Lisp type name, and the logical combinators (`and`, `or`, and `not`) are represented implicitly in the structure of the BDD.

A disadvantage BDDs present when compared to s-expressions as presented in Section 2.5 is the loss of homoiconicity [McI60, Kay69]. Whereas, s-expression based type-specifiers may appear in-line in the Common Lisp code, BDDs may not.

A remarkable fact about this representation is that any two logically equivalent Boolean expressions have exactly the same BDD structural representation, provided the node labels are consistently, totally ordered. Andersen [And99] provides a proof for this claim. For example, the expression from Figure 5.1, $(A_1 \wedge A_2) \vee (A_1 \wedge \neg A_2 \wedge A_3) \vee (\neg A_1 \wedge \neg A_3)$ is equivalent to $\neg((\neg A_1 \vee \neg A_2) \wedge (\neg A_1 \vee A_2 \vee \neg A_3) \wedge (A_1 \vee A_3))$. So they both have the same shape as shown in the Figure 5.1. However, if we naïvely substitute Common Lisp type names for Boolean variables in the BDD representation as suggested by Castagna, we find that this equivalence relation does not hold in many cases related to subtype relations in the Common Lisp type system.

An example is that the Common Lisp two types `(and (not arithmetic-error) array (not base-string))` *vs.* `(and array (not base-string))` are equivalent, but the naïvely constructed BDDs are different:



In order to assure the minimum number of BDD allocations possible, and thus ensure that BDDs which represent equivalent types are actually represented by the same BDD, the suggestion by Andersen [And99] is to intercept the BDD constructor function. This constructor should assure that it never returns two BDD which are semantically equivalent but not `eq`.

7.4 Canonicalization

Several checks are in place to reduce the total number of BDDs allocated, and to help assure that two equivalent Common Lisp types result in the same BDD. The following sections, 7.4.1 through 7.4.5 detail the operations which we found necessary to handle in the BDD construction function in order to assure that equivalent Common Lisp type specifiers result in identical BDDs. The first two come directly from Andersen's work. The remaining are our contribution, and are the cases we found necessary to implement in order to enhance BDDs to be compatible with the Common Lisp type system.

7.4.1 Equal right and left children

An optimization noted by Andersen is that if the left and right children are identical then simply return one of them, without allocating a new BDD [And99].

7.4.2 Caching BDDs

Another optimization noted by Andersen is that whenever a new BDD is allocated, an entry is made into a hash table so that the next time a request is made with the exactly same label, left child, and right child, the already allocated BDD is returned. We associate each new BDD with a unique integer, and create a hash key which is a list (a triple) of the type specifier (the label) followed by two integers corresponding to the left and right children. We use a Common Lisp `equal` hash table for this storage, although we'd like to investigate whether creating a more specific hash function specific to our key might be more efficient.

7.4.3 Reduction in the presence of subtypes

Since the nodes of the BDD represent Common Lisp types, other specific optimizations are made. The cases include situations where types are related to each other in certain ways: subtype, supertype, and disjoint types. In particular, there are 12 optimization cases, detailed in Figure 7.1. Each of these optimizations follows a similar pattern: when constructing a BDD with label X, search in either the left or right child to find a BDD, 

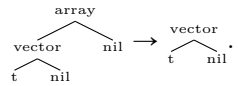
. If X and Y have a particular relation, different for each of the 12 cases, then the 

BDD reduces either to L or R. Two cases, 5 and 7, are further illustrated below.

Case	Child to search	Relation	Reduction	Implication
1	$X.left$	$X \perp Y$	$Y \rightarrow Y.right$	implies 7, 10
2	$X.left$	$X \perp \overline{Y}$	$Y \rightarrow Y.left$	implies 9, 8
3	$X.right$	$\overline{X} \perp Y$	$Y \rightarrow Y.right$	implies 5, 12
4	$X.right$	$\overline{X} \perp \overline{Y}$	$Y \rightarrow Y.left$	implies 6, 11
5	$X.right$	$X \supset Y$	$Y \rightarrow Y.right$	implied by 3
6	$X.right$	$X \supset \overline{Y}$	$Y \rightarrow Y.left$	implied by 4
7	$X.left$	$\overline{X} \supset Y$	$Y \rightarrow Y.right$	implied by 1
8	$X.left$	$\overline{X} \supset \overline{Y}$	$Y \rightarrow Y.left$	implied by 2
9	$X.left$	$X \subset Y$	$Y \rightarrow Y.left$	implied by 2
10	$X.left$	$X \subset \overline{Y}$	$Y \rightarrow Y.right$	implied by 1
11	$X.right$	$\overline{X} \subset Y$	$Y \rightarrow Y.left$	implied by 4
12	$X.right$	$\overline{X} \subset \overline{Y}$	$Y \rightarrow Y.right$	implied by 3

Figure 7.1: BDD optimizations

7.4.4 Reduction to child

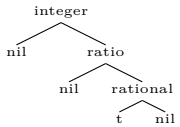
The list of reductions described in Section 7.4.3 fails to apply in cases where the root node itself needs to be eliminated. For example, since $vector \subset array$ we would like the following reductions: 

.

The solution which we have implemented is that before constructing a new BDD, we first ask whether the resulting BDD is type-equivalent to either the left or right child using the `subtypep` function. If so, we simply return the appropriate child without allocating the parent BDD. The expense of this type-equivalence is mitigated by the memoization. Thereafter, the result is in the hash table, and it will be discovered as discussed in Section 7.4.2.

7.4.5 More complex type relations

There are a few more cases which are not covered by the above optimizations. Consider the following BDD:



This represents the type `(and (not integer) (not ratio) rational)`, but in Common Lisp `rational` is identical to `(or integer ratio)`, which means `(and (not integer) (not ratio) rational)` is the empty type. For this reason, as a last resort before allocating a new BDD, we check, using the Common Lisp function `subtypep`, whether the type specifier specifies the `nil` or `t` type. Again this check is expensive, but the expense is mitigated in that the result is cached.

7.4.6 Optimized BDD construction

The BDD constructor takes three arguments: a type specifier (also called a label), and two BDDs called the left and right subtree. Several optimizations are in place to reduce the total number of trees. The most notable optimization is that if the left and right subtrees are identical then simply return one of them, without allocating a new tree [And99].

When the nodes of the BDD represent types, other optimizations can be made. The cases include situations where types are related to each other in certain ways: subtype, supertype, and disjoint types. In particular there are 12 optimization cases, some of which are implied by others. Each of the 12 optimizations follows a similar pattern: when constructing a BDD with label X, search in either the left or right subtree to find a subtree, 

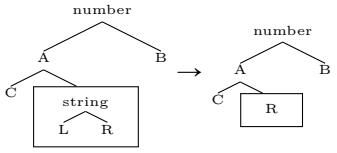
, whose label is Y having left and right subtrees L and R. If X and Y have a particular relation, then the 

tree reduces either to L or R. A summary of the optimizations can be found in Figure 7.1.

Case 1: If $X \cap Y = \varnothing$ and 

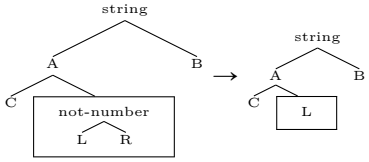
appears in $left(X)$, then 

reduces to R.

For example: If $X = number$ and $Y = string$, we have 

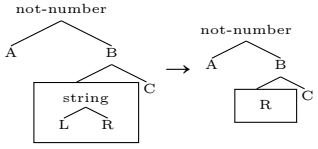
because $(number \cap string = \varnothing)$.

Case 2: If $X \cap \overline{Y} = \varnothing$ and  appears $left(X)$, then  reduces to L.

For example: If $X = string$ and $Y = \overline{number}$, we have 

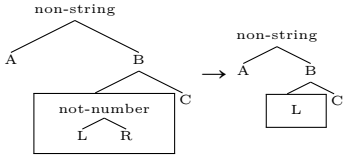
because $string \cap \overline{\overline{number}} = \varnothing$.

Case 3: If $\overline{X} \cap Y = \varnothing$ and  appears $right(X)$, then  reduces to R.

For example: If $X = \overline{number}$ and $Y = string$, we have 

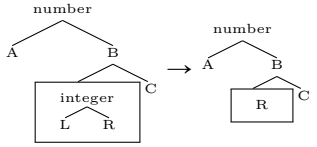
because $\overline{\overline{number}} \cap string = \varnothing$.

Case 4: If $\overline{X} \cap \overline{Y} = \varnothing$ and  appears $right(X)$, then  reduces to L.

For example: If $X = \overline{string}$ and $Y = \overline{number}$, we have 

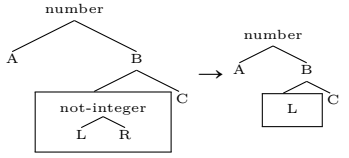
because $\overline{\overline{string}} \cap \overline{\overline{number}} = \varnothing$.

Case 5: If $Y \subset X$ and  appears in $right(X)$, then  reduces to R.

For example: If $X = number$ and $Y = integer$, we have 

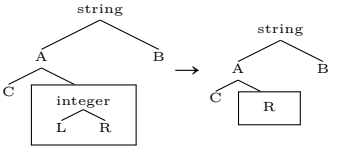
because $integer \subset number$.

Case 6: If $\overline{Y} \subset X$ and  appears in $right(X)$, then  reduces to L.

For example: If $X = number$ and $Y = \overline{integer}$, we have 

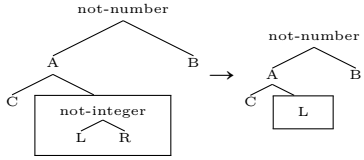
because $\overline{\overline{integer}} \subset number$.

Case 7: If $Y \subset \overline{X}$ and  appears in $left(X)$, then  reduces to R.

For example: If $X = string$ and $Y = integer$, we have 

because $integer \subset \overline{string}$.

Case 8: If $\overline{Y} \subset \overline{X}$ and  appears in $left(X)$, then  reduces to L.

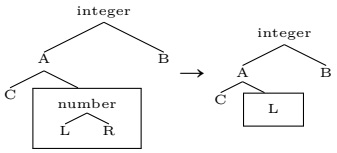
For example: If $X = \overline{integer}$ and $Y = \overline{number}$, we have 

because $\overline{\overline{integer}} \subset \overline{\overline{number}}$.

Case 9: If $X \subset Y$ and 

appears in $left(X)$, then 

reduces to L.

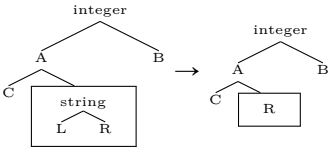
For example: If $X = integer$ and $Y = number$, we have 

because $integer \subset number$.

Case 10: If $X \subset \overline{Y}$ and 

appears in $left(X)$, then 

reduces to R.

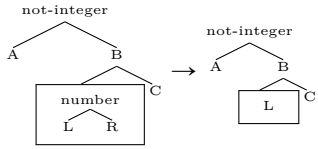
For example: If $X = integer$ and $Y = string$, we have 

because $integer \subset \overline{string}$.

Case 11: If $\overline{X} \subset Y$ and 

appears in $right(X)$, then 

reduces to L.

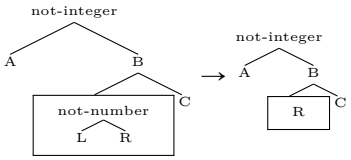
For example: If $X = not - integer$ and $Y = number$, we have 

because $\overline{\overline{integer}} \subset number$.

Case 12: If $\overline{X} \subset \overline{Y}$ and 

appears in $right(X)$, then 

reduces to R.

For example: If $X = \overline{integer}$ and $Y = \overline{number}$, we have 

because $\overline{\overline{integer}} \subset \overline{\overline{number}}$.

7.5 Related work

Newton *et al.* [New17] presented a synopsis of this work at the 2017 European Lisp Symposium.

Maclachlan [Mac92] introduced Python, a Common Lisp compiler. He explains the elimination of `if-if` constructions as part of the ICR (Implicit Continuation Representation) optimization. Maclachlan [Mac03] explains this optimization as equivalent to the transform `(if (if A B C) D E)` ↦ `(if A (if B D E) (if C D E))`. A potential advantage of this transformation is that the blocks `D` and `E` may be optimized differently in the two branches.

On the contrary, a disadvantage is that the transformation grows the compiled code size exponentially, roughly doubling it at each such transformation. The problem would be exacerbated by transformation such as `(not A)` ↦ `(if A nil t)`. The problem of exponential code growth is reminiscent of problem posed by the UOBDD in Section 5.1.1. The solution in the case of UOBDD was to build the decision tree such that this duplication of redundant nodes is avoided by construction.

Charniak *et al.* [CM85] present discrimination nets as a way of formalizing these decision procedures. Charniak notes that the nets may form trees in some cases but does not explore techniques to avoid tree explosion.

PCL (Portable Common Loops) [BKK+86] which is the heart of many implementations of the CLOS (the Common Lisp Object System) [Kee89, Ano87], in particular the implementation within SBCL, uses discrimination nets to optimize generic function dispatch.

7.6 Perspectives

There is a remaining area of research which is of concern. There are exotic cases where two differently structured BDD's may represent the same Common Lisp type; particularly the empty type may take many forms. We are not sure if this problem can be fixed, or whether it is just a loophole in an overly ambitious theory. For a comprehensive theory, we should completely characterize these situations. Our current recommendation is that the "final" types of any computation, still need to be tested to be a subtype of the type `nil` using a call to `subtypep`.

The loophole is something like the following. Assume the types under consideration are `A`, `B`, `C`, `D`, and `E`. Then, for example, if `E` is a subtype of one of subtype of `A`, `B`, `C`, `D`, `(not A)`, `(not B)`, `(not C)`, or `(not D)`,

then the appropriate reduction and uniqueness is preserved in the BDD. But in the case that `E` is a subtype of some Boolean combination, *e.g.*, if $E \subset (A \cup B \cup \overline{C}) \cap (\overline{A} \cup B \cup \overline{D})$, then it may be possible to have multiple, type-equivalent BDDs representing expressions of `E`. We suspect the problem is that if `E` is a subtype of any subtree in the BDD, then we are safe, but if `E` is a subtype of some non-expressed combination, there uniqueness is not guaranteed. *I.e.*, if there is a subtree to eliminate, it can be eliminated, but if the subtype is not expressed in an actual subtree, then Houston we have a problem.

Part III

The Type Decomposition and Serialization Problems

Figure 7.2: Me (center) with my brother and my sister.

In Chapter 4 we presented an elaborate procedure for generating efficient code to recognize regular patterns in heterogeneous sequences. This presentation pointed out two problems which we address in Part III. The type decomposition problem is presented in Chapters 8 with algorithms addressing it in Chapter 9 and the relative performance of those algorithms is analyzed in Chapter 10. Thereafter, we examine the second problem, that of serialization, in Chapter 11.

Chapter 8

Maximal Disjoint Type Decomposition

In Chapter 4 we introduced the problem determinizing a finite state machine whose transitions represent Common Lisp type predicates. The state machine was deterministic only if no two transitions from any given state contained intersecting types. In the current chapter, we look at the theoretical aspects of this problem. We rigorously define the problem and prove that it has a unique solution. However, we do not attempt here to present algorithms to find such a solution; such algorithms are presented in Chapter 9. In Chapter 10 we analyze the performance of these algorithms.

The MDTD problem is that of using Boolean operations to decompose a set of partially overlapping regions into a valid partition. In particular, given $V = \{A_1, A_2, ..., A_M\}$, suppose that for each pair (A_i, A_j), we can ascertain whether one of the following is true: $A_i \subseteq A_j$, $A_i \supseteq A_j$, or $A_i \cap A_j = \varnothing$. We would like to compute the maximal disjoint decomposition of V. We define precisely what we mean by maximal disjoint decomposition in Definition 8.63 of Section 8.2.6.

An illustration should help give an intuition of the problem. The Venn diagram in Figure 8.1 is an example for $V = \{A_1, A_2, ..., A_8\}$. The maximal disjoint decomposition $D = \{X_1, X_2, ..., X_{13}\}$ of V is shown in Figure 8.1 as a Venn diagram, and in Figure 8.3 as a set of Boolean equations. D is the largest possible set of pairwise disjoint subsets of $\{A_1 \cup A_2 \cup ... \cup A_8\}$, for which every element thereof can be expressed as a Boolean combination of elements of V.

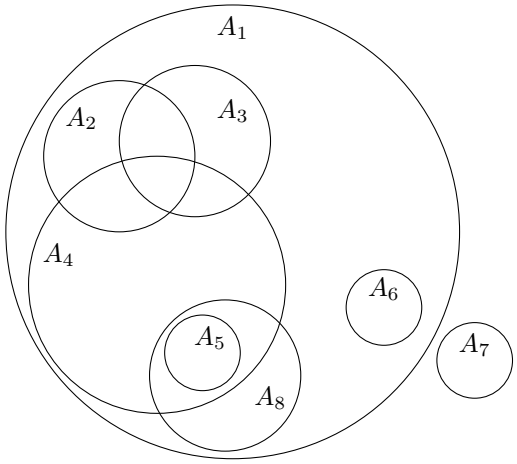


Figure 8.1: Venn Diagram of possibly overlapping sets. Each A_n is represented by a circle on this diagram.

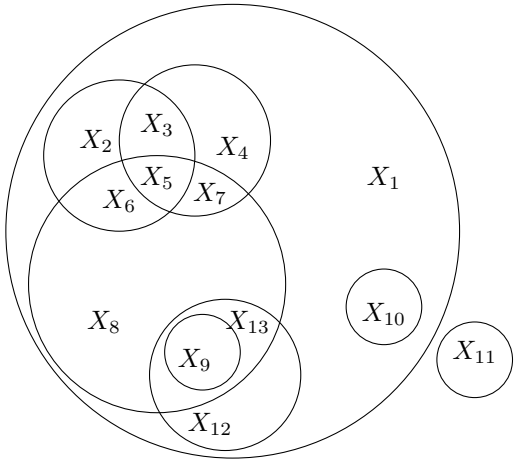


Figure 8.2: Venn Diagram after Decomposition. Each X_i is represented by a region which does not overlay any other region.

Disjoint Set	Derived Expression
X_1	$A_1 \cap \overline{A_2} \cap \overline{A_3} \cap \overline{A_4} \cap \overline{A_6} \cap \overline{A_8}$
X_2	$A_2 \cap \overline{A_3} \cap \overline{A_4}$
X_3	$A_2 \cap A_3 \cap \overline{A_4}$
X_4	$A_3 \cap \overline{A_2} \cap \overline{A_4}$
X_5	$A_2 \cap A_3 \cap A_4$
X_6	$A_2 \cap A_4 \cap \overline{A_3}$
X_7	$A_3 \cap A_4 \cap \overline{A_2}$
X_8	$A_4 \cap \overline{A_2} \cap \overline{A_3} \cap \overline{A_8}$
X_9	A_5
X_{10}	A_6
X_{11}	A_7
X_{12}	$A_8 \cap \overline{A_4}$
X_{13}	$A_4 \cap A_8 \cap \overline{A_5}$

Figure 8.3: Decomposition of Venn diagram in Figure 8.1. Each X_i is expressed as a Boolean combination of elements of $\{A_1, ..., A_n\}$.

The details in this chapter may not be useful to the casual reader. We invite such a reader to review Section 8.1 which introduces the problem via illustrations, and then skip to Section 8.2.6 which states and proves the claims of uniqueness.

8.1 Motivation

Newton *et al.* [NDV16] presented this problem when attempting to determinize automata used to recognize rational type expressions. It was stated in that work that the algorithm employed there had performance issues which needed to be addressed. Newton *et al.* showed such performance improvements in a follow-up paper [NVC17].

Another potential application of this problem, which is still an open area of active research, is the problem of re-ordering clauses in a `typecase` in Common Lisp [Ans94], or similar constructs in other programming languages. The property to note is that, given an expression such as in Example 8.1,

Example 8.1 (`typecase` with independent clauses)**.**

```
(typecase object
  (number ...)
  (symbol ...)
  (array ...)
  ...)
```

the clauses can be freely reordered, provided the types are disjoint. Re-ordering the clauses is potentially advantageous for computational efficiency if the type which is most likely to occur at run-time appears first. Another reason to reorder is so that types can be simplified. Consider Example 8.2

Example 8.2 (`typecase` whose clauses contain disjoint types)**.**

```
(typecase object
  ((and number (not integer)) E1)
  (integer E2)
  (array E3)
  ...)
```

The clauses of the `typecase` in Example 8.2 cannot be simplified as presented. However, we are allowed to swap the first and second clause the `typecase` because the two types, `(and number (not integer))` and `integer` are disjoint. After swapping the clauses and simplifying, we end up with code as in Example 8.3.

Example 8.3 (`typecase` after re-ordering and simplification)**.**

```
(typecase object
  (integer E2)
  (number E1)
  (array E3)
  ...)
```

Thus, it may be interesting to compute a minimal disjoint type decomposition in order to express intermediate forms, which are thereafter simplified to more efficient carefully ordered clauses of intersecting types. This topic is further discussed in Chapter 11.

Finally, still another reason for studying this problem is because it allows us to examine lower level algorithms and data structures, the specifics of which may themselves have consequences which outweigh the type disjunction problem itself. There are several such cases in this technical report: including techniques and practices for using BDDs to represent type specifiers (Section 9.5) and techniques for optimizing programs characterized by heavy use of `subtypep` (Chapter 10).

8.2 Rigorous development

In this section, we define exactly what we mean by the term *maximal disjoint decomposition*. The main result of this section is Theorem 8.67 which claims the existence and uniqueness of the maximal disjoint decomposition. The reader who does not care about the rigorous treatment may skip most of this section, as long as he grasps the definition of *maximal disjoint decomposition* (Definition 8.63) and the claims of its existence and uniqueness (Theorem 8.67).

The presentation order used in this section is bottom-up; *i.e.*, we attempt to define and prove everything needed before it is actually used. This order may cause some difficulty to the reader, as it may not be clear at each point why something is being introduced. We attempt to alleviate some of this problem by providing motivational discussions as prelude to each section and by giving sufficiently many examples, so that even if the reader does not foresee how something will be used later, exactly why it is being presented, at least the reader can get a rigorous definition but also an intuitive feeling of the concept.

8.2.1 Unary set operations

Notation 8.4. We denote the cardinality of set A, *i.e.* the number of elements, by $|A|$. We say that an infinite set has infinite cardinality; otherwise it has finite cardinality.

Example 8.5 (Cardinality of the empty set)**.** $|\varnothing| = 0$.

Example 8.6 (Misleading cardinality of set denoted by variables)**.** If $A = \{X, Y\}$, then $1 \leqslant |A| \leqslant 2$; because A is not empty, and it might be that $X = Y$.

Definition 8.7. If V is a set of subsets of U, then we define the unary $\bigcup$ operator as follows:

$$\bigcup V = \begin{cases} \varnothing & \text{if } V = \varnothing \\ X & \text{if } |V| = 1 \text{ and } V = \{X\} \\ \bigcup\limits_{X \in V} X & \text{if } |V| > 1 \end{cases}$$

Definition 8.8. If V is a set of subsets of U, then we define the unary $\bigcap$ operator as follows:

$$\bigcap V = \begin{cases} U & \text{if } V = \varnothing \\ X & \text{if } |V| = 1 \text{ and } V = \{X\} \\ \bigcap\limits_{X \in V} X & \text{if } |V| > 1 \end{cases}$$

Example 8.9 (Unary intersection and union operations)**.** Let $V = \{\{1\}, \{1,2\}, \{1,2,4\}, \{1,3,4,5\}\}$, then

$$\bigcap V = \{1\} \cap \{1,2\} \cap \{1,2,4\} \cap \{1,3,4,5\} = \{1\}$$

and

$$\bigcup V = \{1\} \cup \{1,2\} \cup \{1,2,4\} \cup \{1,3,4,5\} = \{1,2,3,4,5\}$$

Example 8.10 (Unary union of singleton set)**.** Let $X = \{12, 13, 14\}$, and let $V = \{X\} = \{\{12, 13, 14\}\}$, as in the middle case of Definition 8.7. In this case $|V| = 1$, and $|\bigcup V| = 3$, because

$$\bigcup V = \bigcup \{X\} = X = \{12, 13, 14\}\,.$$

Moreover, if $V = \{\{X\}\} = \{\{\{12, 13, 14\}\}\}$, then $|V| = 1$ and $|\bigcup V| = 1$, because

$$\bigcup V = \bigcup \{\{X\}\} = \{X\} = \{\{12, 13, 14\}\}\,.$$

The values of $\bigcup V$ and $\bigcap V$ in the cases where $|V| = 0$ and $|V| = 1$ are defined as such so that the notation is consistent and intuitive. In particular, the following identities hold:

$$\bigcup (V \cup V') = (\bigcup V) \cup (\bigcup V')$$
$$\bigcap (V \cup V') = (\bigcap V) \cap (\bigcap V')$$

Note that the definitions of $\bigcup V$ and $\bigcap V$ in no way make a claim or supposition about the cardinality of V. We may use the same notation whether V is infinite or finite.

Definition 8.11. By $\mathbb{P}(U)$ we denote the power set of U, *i.e.* the set of subsets of U. Consequently we may take $V \subseteq \mathbb{P}(U)$ to mean that V is a set of subsets, each begin a subset of U.

Example 8.12 (Power set)**.** If $U = \{1, 2, 3\}$, then $\mathbb{P}(U) = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$. Notice that $|U| = 3$ and $|\mathbb{P}(U)| = 2^3 = 8$. It holds in general that $|\mathbb{P}(U)| = 2^{|U|}$.

8.2.2 Partitions and covers

In this report, we often refer to the relation of *disjoint-ness* or *intersection* between two sets. For this reason, we introduce a notation which some readers may find non-standard. We remind the reader of Notations 2.2 and 2.3 for denoting the disjoint and non-disjoint relations.

Definition 8.13. Let $D = \{X_1, X_2, ..., X_M\}$, with $X_i \subseteq U$ for $1 \leqslant i \leqslant M$. If

$$X_i \neq X_j \implies X_i \parallel X_j \,,$$

then D is said to be disjoined in U. *I.e.*, a set disjoined in a given U is a set of mutually disjoint subsets of U.

Example 8.14 (Disjoined set and non-disjoined sets)**.** The set $\{\{1,3,5\},\{0,2\},\{4,6\},\varnothing\}$ is disjoined because its elements are sets, none of which have a common element. By contrast, the set $\{\{1,2,3\},\{2,4\},\{5\}\}$ is not disjoined as 2 is common to $\{1,2,3\}$ and $\{2,4\}$.

Note that if a set V is disjoined then $\bigcap V = \varnothing$. However, from Example 8.14 we see that $\bigcap V = \varnothing$ does not imply that V is disjoined.

Definition 8.15. If $V \subseteq \mathbb{P}(U)$ and $C \subseteq \mathbb{P}(U)$, C is said to be a cover of V, or equivalently we say that C covers V, provided $\bigcup V \subseteq \bigcup C$. Furthermore, if $\bigcup V = \bigcup C$ we say that C is an exact cover of V or that C exactly covers V.

Example 8.16 (Sets as covers)**.** If

$$U = \{\{1,2,3\},\{3,4\}\}$$

and

$$V = \{\{1,2,3\},\{2,4,6\},\varnothing\}\,,$$

then V covers U, because

$$\bigcup U \subset \bigcup V\,;$$

in particular

and

$$\bigcup U = \bigcup\{\{1,2,3\},\{3,4\}\} = \{1,2,3\} \cup \{3,4\} = \{1,2,3,4\} = \{1,2,3\}$$
$$\bigcup V = \bigcup\{\{1,2,3\},\{2,4,6\},\varnothing\} = \{1,2,3\} \cup \{2,4,6\} \cup \varnothing = \{1,2,3,4,6\}\,.$$

And

$$\{1,2,3,4\} \subset \{1,2,3,4,6\}\,.$$

However, V is not an exact cover of U because $6 \in \bigcup V$ but $6 \notin \bigcup U$, *i.e.* $\bigcup V \nsubseteq U$.

Definition 8.17. A partition of a set V is a disjoined set P with the property that $\bigcup P = V$. Consequently, a disjoined set may be said to partition its union.

Example 8.18 (A set which is a partition of a given set)**.** The disjoined set $D = \{\{0\}, \{2,4,6,8\}, \{1,3,5,7,9\}\}$ is a partition of $\{0,1,2,3,4,5,6,7,8,9\}$, because the three sets (the elements of D) are mutually disjoint and

$$\bigcup D = \{0\} \cup \{2,4,6,8\} \cup \{1,3,5,7,9\} = \{0,1,2,3,4,5,6,7,8,9\}$$

Example 8.19 (A set which is not a partition of a given set)**.** The set $V = \{\{1\}, \{1,2\}, \{2,4\}, \{3,4,5\}\}$ from Example 8.9 is not a partition of $\{1,2,3,4,5\}$ because V is not disjoined; *i.e.*, the elements of V are not mutually disjoint. However, V does cover $\{1,2,3,4,5\}$.

8.2.3 Sigma algebras

The purpose of the next several definitions is to define the set $\mathscr{B}(V)$ which is intended to be the set of all Boolean combinations of sets in a given set V. $\mathscr{B}(V)$ is defined in Definition 8.29, and is proved to exist and be unique in Corollary 8.30.

The following lemma (Lemma 8.20) will be used in Example 8.26.

Lemma 8.20. *If V is a set of subsets of U, and $\exists X_0 \in V$ such that $X \in V \implies X_0 \subseteq X$, then $X_0 = \bigcap V$.*

Proof. $X \in V \implies X_0 \subseteq X$, therefore $X_0 \subseteq \bigcap V$.

Case 1: $(\bigcap V) \backslash X_0 = \varnothing$, therefore $\bigcap V \subseteq X_0$

Case 2: $(\bigcap V) \backslash X_0 \neq \varnothing$, so let $\alpha \in (\bigcap V) \backslash X_0$. This means $\alpha \notin X_0$, rather $\alpha \in \bigcap V$. $\alpha \in \bigcap V$ means that $\forall X \in V, \alpha \in X$, which is a contradiction because $X_0 \in V$.

□

Intuitively, Lemma 8.20 says that given a set of subsets, if one of those subsets happens to be a subset of all the given subsets, then it is in fact the intersection of all the subsets.

Example 8.21 (Example of Lemma 8.20)**.** Let $V = \{\varnothing, \{1\}, \{2\}\}$. Notice that there is an X_0, namely $X_0 = \varnothing$, which has the property that it is a subset of every element of V, *i.e.*, $X \in V \implies X_0 \subseteq X$. Therefore, $\bigcap V = X_0 = \varnothing$.

Example 8.22 (Another example of Lemma 8.20)**.** Let $V = \{\{1\}, \{1,2\}, \{1,2,3\}, \{1,2,3,4\}, ..., \{1,2,3,4,...,N\}\}$ for some N. Notice that there is an X_0, namely $X_0 = \{1\}$, which has the property that it is a subset of every element of V, *i.e.*, $\{1\}$ is a subset of every element of V, or $X \in V \implies X_0 \subseteq X$. Therefore, $\bigcap V = \{1\}$.

Definition 8.23. Let $V \subseteq U$, and let F be a set of binary functions mapping $U \times U \mapsto U$. A superset $V' \supseteq V$ is said to be a closed superset of V under F if $\alpha, \beta \in V'$ and $f \in F \implies f(\alpha, \beta) \in V'$.

Example 8.24 (Closed superset)**.** If $V = \{1\}$, and $F = \{+\}$, then the set of integers $\mathbb{Z}$ is a closed superset of V under F. Why, because $V \subseteq \mathbb{Z}$, and $\alpha, \beta \in \mathbb{Z} \implies \alpha + \beta \in \mathbb{Z}$.

Definition 8.25. Let $V \subseteq U$, and let F be a set of binary functions mapping $U \times U \mapsto U$. The closure of V under F, denoted $clos_F(V)$, is the intersection of all closed supersets of V under F.

Example 8.26 (Example of $clos_F(V)$)**.** If $V = \{1\}$, and $F = \{+\}$, then $clos_F(V)$ is the set of positive integers, $\mathbb{N}$.

Proof. To show this we argue that $\mathbb{N}$ is a closed superset of V and that if V' is a closed superset of V then $\mathbb{N} \subseteq V'$.

$1 \in \mathbb{N}$ so $V \subseteq \mathbb{N}$. If $\alpha, \beta \in \mathbb{N}$, then $\alpha + \beta \in \mathbb{N}$, so $\mathbb{N}$ is closed.

Let V' be a closed superset of V. $\mathbb{N} \subseteq V'$ can be shown by induction. $1 \in V'$ because $1 \in V \subseteq V'$. Now assume $k \in V'$. Is $k + 1 \in V'$? Yes, because $\{k\} \cup \{1\} = \{k, 1\} \subseteq V' \implies k + 1 \in V'$. Therefore $\mathbb{N} \subseteq V'$.

So by Lemma 8.20, $\mathbb{N} = clos_F(V)$. □

In Definition 8.25 we defined something called the closure, but we need to prove that it is unique and closed, thus deserving of its name. In Theorem 8.27 we argue the existence and uniqueness of this so-called closure, and also argue that it is indeed closed according to Definition 8.23.

Theorem 8.27. *If $V \subseteq U$, and if F is a set of binary functions defined on U, then there exists a unique $clos_F(V)$, and it is closed under F.*

Proof. First we show, by construction, that at least one closed superset of V exists. Define a monotonic sequence $\{\Phi_n\}_{n=0}^{\infty}$ of sets as follows:

- $\Phi_0 = V$
- If $i > 0$, then $\Phi_i = \Phi_{i-1} \cup \bigcup_{f \in F} \{f(x, y) \mid x, y \in \Phi_{i-1}\}$

By construction $\Phi_i \subseteq \Phi_{i+1}$. Define the set $\Phi = \bigcup_{i=0}^{\infty} \Phi_i$. We know that $V = \Phi_0 \subseteq \bigcup_{i=0}^{\infty} \Phi_i$. Next, let $\alpha \in \Phi$, $\beta \in \Phi$, $f \in F$; take $n \geqslant 0$ such that $\alpha, \beta \in \Phi_n$. By definition $f(\alpha, \beta) \in \Phi_{n+1} \subseteq \Phi$. Thus Φ is closed under F.

Now that we know at least one such closed set exists, suppose Ψ is the set of all supersets of V which are closed under F. Let $\Phi = \bigcap \Psi$. Φ is uniquely defined because intersection is well defined even on infinite sets. But the question remains whether Φ is closed under F.

We now show that Φ satisfies the definition of closed under F. V is a subset of every element of Ψ so $V \subseteq \bigcap \Psi = \Phi$. Now, take $\alpha, \beta \in \Phi$, and $f \in F$. Since $\alpha \in \Phi$, that means α is in every element of Ψ, similarly for β, so $f(\alpha, \beta)$ is every element of Ψ, which means $f(\alpha, \beta) \in \bigcap \Psi = \Phi$. *I.e.* $\Phi = clos_F(V)$ is closed under F. □

Example 8.28 (Closure of set operations)**.** Let $V = \{\{1, 2\}, \{2, 3\}\}$, and F be the set containing the set-union and set-intersection binary operations, denoted $F = \{\cup, \cap\}$. Then $clos_F(V) = \{\varnothing, \{1, 2\}, \{2\}, \{2, 3\}, \{1, 2, 3\}\}$, because if we take $\alpha, \beta \in \{\varnothing, \{1, 2\}, \{2\}, \{2, 3\}, \{1, 2, 3\}\}$, then both $\alpha \cup \beta$ and $\alpha \cap \beta$ are also therein. This can be verified exhaustively as in Figure 8.4.

Any smaller set would not fulfill the definition of $clos_F(V)$, which can also be verified exhaustively. In particular, if either or $\{1, 2\}$ or $\{2, 3\}$ were omitted, then it would no longer be a superset of V, and if any of $\{\varnothing, \{2\}, \{1, 2, 3\}\}$ were omitted, then it would no longer be a closed superset of V. Finally, if any element were added, such as $V' = V \cup \{3\}$, it would no longer fulfill the intersection requirement; *i.e.* $V' \not\subseteq V$ so V' is not the intersection of all closed supersets of V under F.

Definition 8.29. If $V \subseteq \mathbb{P}(U)$, and F is the set of three primitive set operations union, intersection, and relative complement, ($F = \{\cup, \cap, \setminus\}$) then we denote $clos_F(V)$ simply by $\mathscr{B}(V)$ and call it the sigma algebra of V. Moreover, each element of $\mathscr{B}(V)$ is called a Boolean combination of elements of V.

α	β	$\alpha \cup \beta$	$\alpha \cap \beta$
$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
$\varnothing$	$\{1,2\}$	$\{1,2\}$	$\varnothing$
$\varnothing$	$\{2\}$	$\{2\}$	$\varnothing$
$\varnothing$	$\{2,3\}$	$\{2,3\}$	$\varnothing$
$\varnothing$	$\{1,2,3\}$	$\{1,2,3\}$	$\varnothing$
$\{1,2\}$	$\varnothing$	$\{1,2\}$	$\varnothing$
$\{1,2\}$	$\{1,2\}$	$\{1,2\}$	$\{1,2\}$
$\{1,2\}$	$\{2\}$	$\{1,2\}$	$\{2\}$
$\{1,2\}$	$\{2,3\}$	$\{1,2,3\}$	$\{2\}$
$\{1,2\}$	$\{1,2,3\}$	$\{1,2,3\}$	$\{1,2\}$
$\{2\}$	$\varnothing$	$\{2\}$	$\varnothing$
$\{2\}$	$\{1,2\}$	$\{1,2\}$	$\{2\}$
$\{2\}$	$\{2\}$	$\{2\}$	$\{2\}$
$\{2\}$	$\{2,3\}$	$\{2,3\}$	$\{2\}$
$\{2\}$	$\{1,2,3\}$	$\{1,2,3\}$	$\{2\}$
$\{2,3\}$	$\varnothing$	$\{2,3\}$	$\varnothing$
$\{2,3\}$	$\{1,2\}$	$\{1,2,3\}$	$\{2\}$
$\{2,3\}$	$\{2\}$	$\{2,3\}$	$\{2\}$
$\{2,3\}$	$\{2,3\}$	$\{2,3\}$	$\{2,3\}$
$\{2,3\}$	$\{1,2,3\}$	$\{1,2,3\}$	$\{2,3\}$
$\{1,2,3\}$	$\varnothing$	$\{1,2,3\}$	$\varnothing$
$\{1,2,3\}$	$\{1,2\}$	$\{1,2,3\}$	$\{1,2\}$
$\{1,2,3\}$	$\{2\}$	$\{1,2,3\}$	$\{2\}$
$\{1,2,3\}$	$\{2,3\}$	$\{1,2,3\}$	$\{2,3\}$
$\{1,2,3\}$	$\{1,2,3\}$	$\{1,2,3\}$	$\{1,2,3\}$

Figure 8.4: Closure under a set of operations

Corollary 8.30. *If $V \subseteq \mathbb{P}(U)$, $\mathscr{B}(V)$ exists and is unique.*

Proof. Simple application of Theorem 8.27. □

Example 8.31 (A Sigma algebra)**.** Let $V = \{\{1,2\},\{2,3\}\}$ as in Example 8.28.
$\mathscr{B}(V) = \{\varnothing,\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}$. *I.e.*, $\mathscr{B}(V) = \mathbb{P}(\{1,2,3\})$.

8.2.4 Finitely many Boolean combinations

In this section, we provide a tedious proof which was used in the proofs of Corollary 8.54 and Corollary 8.55. In particular, the main result, Theorem 8.51, is the claim that given a finite set V of sets, then the set of all Boolean combinations of these sets is itself finite.

The reader may wish to skip the details of this section, and just concentrate on the main result, Theorem 8.51.

Definition 8.32. Given a finite set of sets $V = \{D_1, D_2, ..., D_n\}$, the conjunctive closure of V, denoted $\mathfrak{C}(V)$, is defined as follows:

$$\mathfrak{C}(V) = \{\bigcap S \mid S \subset V\}.$$

Definition 8.33. Given a finite set of sets $V = \{D_1, D_2, ..., D_n\}$, the disjunctive closure of V, denoted $\mathfrak{D}(V)$, is defined as follows:

$$\mathfrak{D}(V) = \{\bigcup S \mid S \subset V\}.$$

An equivalent way of thinking about definitions 8.32 and 8.33 is that if $S \subset V$, then $\bigcap S \in \mathfrak{C}(V)$ and $\bigcup S \in \mathfrak{D}(V)$. Consequently, if S' is a collection of subsets of V, *i.e.*, $S' \subset \mathbb{P}(V)$, then $\bigcap\bigcup S' \in \mathfrak{D}(\mathfrak{C}(S'))$.

Definition 8.34. Given a finite set of sets V, the distributed complement of V, denoted $\widetilde{V}$ is defined as

$$\widetilde{V} = \{(\bigcup V)\backslash\delta \mid \delta \in V\}.$$

Lemma 8.35 and 8.36 claim that $\mathfrak{C}(V)$ and $\mathfrak{D}(V)$ are both closed, and thus deserving of the names conjunctive closure and disjunctive closure. The proofs follow immediately from definitions 8.32 and 8.33 respectively.

Lemma 8.35. *If V is a finite set, then $\mathfrak{C}(V)$ is closed under intersection.*

Proof. Let $a, b \in \mathfrak{C}(V)$. There exist $A, B \subset V$, such that $a = \bigcap A$ and $b = \bigcap B$. Since A and B are both finite:

$$a \cap b = (\bigcap A) \cap (\bigcap B) = \bigcap(A \cup B)$$

but $A \cup B \subset V$, so $\bigcap(A \cup B) \in \mathfrak{C}(V)$. □

Lemma 8.36. *If V is a finite set, then $\mathfrak{D}(V)$ is closed under union.*

Proof. Let $a, b \in \mathfrak{D}(V)$. There exist $A, B \subset V$, such that $a = \bigcup A$ and $b = \bigcup B$. Since A and B are both finite:

$$a \cup b = (\bigcup A) \cup (\bigcup B) = \bigcup(A \cup B)$$

but $A \cup B \subset V$, so $\bigcup(A \cup B) \in \mathfrak{D}(V)$. □

Example 8.37 (Examples of distributed complement and other sets)**.** Let $V = \{\{1\}, \{1,2\}, \{2,3\}\}$.

$$\begin{aligned}
\bigcup V &= \{1,2,3\} \\
\mathbb{P}(\cup V) &= \{\varnothing, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\} \\
\mathscr{B}(V) &= \{\varnothing, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\} \\
\mathbb{P}(V) &= \{\varnothing, \{\{1\}\}, \{\{1,2\}\}, \{\{2,3\}\}, \\
&\qquad \{\{1\},\{1,2\}\}, \{\{1\},\{2,3\}\}, \{\{1,2\},\{2,3\}\}, \\
&\qquad \{\{1\},\{1,2\},\{2,3\}\}\} \\
\widetilde{V} &= \{\{1,2,3\}\backslash\{1\}, \{1,2,3\}\backslash\{1,2\}, \{1,2,3\}\backslash\{2,3\}\} \\
&= \{\{2,3\}, \{3\}, \{1\}\}
\end{aligned}$$

In this case $\mathscr{B}(V) = \mathbb{P}(\cup V)$, but that is not true in general. Take for example $V = \{\{1,2\}\}$, in which case $\mathscr{B}(V) = \{\varnothing, \{1,2\}\}$, but $\mathbb{P}(\cup V) = \{\varnothing, \{1\}, \{2\}, \{1,2\}\}$.

Example 8.38 (Calculating the conjunctive closure of a set)**.** To calculate $\mathfrak{C}(V)$ we must calculate the intersection of each subset of V, *i.e.* to each element of $\mathbb{P}(V)$. Some of these intersections are redundant; *e.g.*, $\bigcap\{\{1\}\} = \bigcap\{\{1\},\{1,2\}\} = \{1\}$.

$$\begin{aligned}\mathfrak{C}(V) = \{&\bigcap\varnothing, \bigcap\{\{1\}\}, \bigcap\{\{1,2\}\}, \bigcap\{\{2,3\}\},\\ &\bigcap\{\{1\},\{1,2\}\}, \bigcap\{\{1\},\{2,3\}\}, \bigcap\{\{1,2\},\{2,3\}\},\\ &\bigcap\{\{1\},\{1,2\},\{2,3\}\}\}\\ = \{&\varnothing,\{1\},\{1,2\},\{2,3\},\{1\},\varnothing,\{2\},\varnothing\}\\ = \{&\varnothing,\{1\},\{2\},\{1,2\},\{2,3\}\}\end{aligned}$$

Lemma 8.41 predicts that $|\mathfrak{C}(V)| \leqslant 2^{|V|}$, which is true as $5 \leqslant 2^3$. In addition, Lemma 8.35 predictions that $\mathfrak{C}(V)$ is closed under intersection. We see that this is the case; *e.g.*,

$$\bigcap\{\{2\},\{1,2\},\{1,2,3\}\} = \{\{2\} \cap \{1,2\} \cap \{1,2,3\}\} = \{2\} \in \mathfrak{C}(V)\,.$$

Example 8.39 (Calculating the disjunctive closure of a set)**.** To calculate $\mathfrak{D}(V)$ we must calculate the union of each subset of V, *i.e.* to each element of $\mathbb{P}(V)$. Some of these unions are redundant; *e.g.*, $\bigcap\{\{1,2\},\{3\}\} = \bigcap\{\{1,2\},\{2,3\}\} = \{1,2,3\}$.

$$\begin{aligned}\mathfrak{D}(V) = \{&\bigcup\varnothing, \bigcup\{\{1\}\}, \bigcup\{\{1,2\}\}, \bigcup\{\{2,3\}\},\\ &\bigcup\{\{1\},\{1,2\}\}, \bigcup\{\{1\},\{2,3\}\}, \bigcup\{\{1,2\},\{2,3\}\},\\ &\bigcup\{\{1\},\{1,2\},\{2,3\}\}\}\\ = \{&\varnothing,\{1\},\{1,2\},\{2,3\},\{1,2\},\{1,2,3\},\{1,2,3\},\{1,2,3\}\}\\ = \{&\varnothing,\{1\},\{1,2\},\{2,3\},\{1,2,3\}\}\end{aligned}$$

Example 8.40 (Calculating the disjunctive closure of a conjunctive closure)**.** To calculate $\mathfrak{D}(\mathfrak{C}(V))$ we must calculate the union of each subset of $\mathfrak{C}(V)$. There are $2^5 = 32$ subsets of $\mathfrak{C}(V)$, but when calculating their union we find many redundancies.

$$\mathfrak{D}(\mathfrak{C}(V)) = \{\varnothing,\{1\},\{2\},\{1,2\},\{2,3\},\{1,2,3\}\}$$

We see that $\mathfrak{D}(\mathfrak{C}(V)) = \mathbb{P}(V) \setminus \{\{3\},\{1,3\}\}$, so in this case $\mathscr{B}(V) \not\subset \mathfrak{D}(\mathfrak{C}(V))$. However, if we see that $\mathscr{B}(V) \subset \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$ as predicted by Lemma 8.50. We also see that $\widetilde{V} \neq \overline{V}$.

$$\begin{aligned}V \cup \widetilde{V} &= \{\{1\},\{1,2\},\{2,3\}\} \cup \{\{2,3\},\{3\},\{1\}\}\\ &= \{\{1\},\{3\},\{1,2\},\{2,3\}\}\\ \mathfrak{C}(V \cup \widetilde{V}) &= \{\varnothing,\{1\},\{2\},\{3\},\{1,2\},\{2,3\}\}\\ \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V})) &= \{\varnothing,\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}\end{aligned}$$

As a coincidence, in this case $\mathscr{B}(V) = \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V})) = \mathbb{P}(\cup V)$, but that is not true in general. Take for example $V = \{\{1,2\}\}$, in which case $\mathscr{B}(V) = \{\varnothing,\{1,2\}\}$, but $\mathbb{P}(\cup V) = \mathbb{P}(V) = \{\varnothing,\{1\},\{2\},\{1,2\}\}$.

Lemma 8.41. *If V is a finite set of sets, there are at most $2^{|V|}$ elements of $\mathfrak{C}(V)$, i.e.,*

$$|\mathfrak{C}(V)| \leqslant 2^{|V|}\,.$$

Proof. Each element of $\mathfrak{C}(V)$ is the intersection of the elements of some element of the power set of V, *i.e.* if $m \in \mathfrak{C}(V)$ then there exists $S \subset V$ or $S \in \mathbb{P}(V)$, such that $m = \bigcap S$. $|\mathbb{P}(V)| = 2^{|V|}$. Therefore

$$|\mathfrak{C}(V)| \leqslant 2^{|V|}.$$

□

Lemma 8.42. *If V is a finite set of sets, there are at most $2^{|V|}$ elements of $\mathfrak{D}(V)$, i.e.,*

$$|\mathfrak{D}(V)| \leqslant 2^{|V|}.$$

Proof. Each element of $\mathfrak{D}(V)$ is the union of the elements of some element of the power set of V, *i.e.* if $M \in \mathfrak{D}(V)$ then there exists $S \subset V$ or $S \in \mathbb{P}(V)$, such that $M = \bigcup S$. $|\mathbb{P}(V)| = 2^{|V|}$. Therefore

$$|\mathfrak{D}(V)| \leqslant 2^{|V|}.$$

□

Definition 8.43. The distributed intersection of a set of sets is defined as follows. Let V be a set of subsets of G; *i.e.*, $V \subset \mathbb{P}(G)$. Let $\mathfrak{F}\colon \mathbb{P}(\mathbb{P}(V)) \to \mathbb{P}(G)$ be defined as follows:

$$\mathfrak{F}(S) = \bigcup \{\cap s \mid s \in S\}$$

Example 8.44 (Calculating a distributed intersection)**.** Let V and S be defined as follows:

$$\begin{aligned} V &= \{\{1\},\{2\},\{1,2\},\{2,3\},\{2,4\}\} \\ S_1 &= \{\{1\},\{1,2\},\{1,3\}\} \\ S_2 &= \{\{2\},\{1,2\}\} \\ S &= \{S_1, S_2\} \\ &= \{\{\{1\},\{1,2\},\{1,3\}\},\{\{2\},\{1,2\}\}\}. \end{aligned}$$

Then,

$$\begin{aligned} \mathfrak{F}(S) &= \bigcup \{\cap s_i \mid s_i \in S\} \\ &= \bigcup \{\, \cap s_i \mid s_i \in \{\{\{1\},\{1,2\},\{1,3\}\},\{\{2\},\{1,2\}\}\}\} \\ &= \bigcup \{\quad \cap \{\{1\},\{1,2\},\{1,3\}\},\quad \cap \{\{2\},\{1,2\}\} \quad\} \\ &= \bigcup \{\quad \{1\} \cap \{1,2\} \cap \{1,3\},\quad \{2\} \cap \{1,2\} \quad\} \\ &= \bigcup \{\{1\},\{2\}\} \\ &= \{1\} \cup \{2\} \\ &= \{1,2\} \end{aligned}$$

One might be tempted to wonder whether $\mathfrak{F}(S) \stackrel{?}{=} \bigcup\bigcap S$, but it is not in general. In the case where S is defined as above, the two are equal.

$$\begin{aligned} \bigcup\bigcap S &= \bigcup\bigcap \{S_1, S_2\} \\ &= \bigcup (\bigcap \{S_1, S_2\}) \\ &= \bigcup (S_1 \cap S_2) \\ &= \bigcup \{\{1,2\}\} \\ &= \{1,2\} \end{aligned}$$

However, if $S_3 = \{\{2\}, \{1,2,3\}\}$, then we see that $\mathfrak{F}(\{S_1, S_3\}) \neq \bigcup\bigcap\{S_1, S_3\}$.

$$\begin{aligned}
\mathfrak{F}(\{S_1, S_3\}) &= \bigcup\{ \quad \cap\{\{1\},\{1,2\},\{1,3\}\}, \quad \cap\{\{2\},\{1,2,3\}\} \quad \} \\
&= \bigcup\{ \quad \{1\} \cap \{1,2\} \cap \{1,3\}, \quad \{2\} \cap \{1,2,3\} \quad \} \\
&= \bigcup\{\{1\},\{2\}\} \\
&= \{1\} \cup \{2\} \\
&= \{1,2\}
\end{aligned}$$

$$\begin{aligned}
\bigcup\bigcap\{S_1, S_3\} &= \bigcup(\bigcap\{S_1, S_3\}) \\
&= \bigcup(S_1 \cap S_3) \\
&= \bigcup\left(\{\{1\},\{1,2\},\{1,3\}\} \cap \{\{2\},\{1,2,3\}\}\right) \\
&= \bigcup \varnothing \\
&= \varnothing
\end{aligned}$$

Corollaries 8.45 and 8.45 follow immediately from the definitions $\mathfrak{C}$, $\mathfrak{D}$, and $\mathfrak{F}$. Nevertheless, they are presented because they help simplify notation in the proof of Lemmas 8.47, 8.48, and 8.49.

Corollary 8.45. *If V is a set of sets, and $S \subset \mathbb{P}(V)$, then $\mathfrak{F}(S) \in \mathfrak{D}(\mathfrak{C}(V))$.*

Proof. $\mathfrak{F}(S)$ is a union of the intersections of particular subsets of $\mathbb{P}(V)$. $\mathfrak{D}(\mathfrak{C}(V))$ is the set of all such unions. Therefore $\mathfrak{F}(S) \in \mathfrak{D}(\mathfrak{C}(V))$. □

Corollary 8.46. *If V is a set of sets, and if $x \in \mathfrak{D}(\mathfrak{C}(V))$, then there exists an $S \in \mathbb{P}(\mathbb{P}(V))$, not necessarily unique, such that $\mathfrak{F}(S) = x$.*

Proof. As it is defined, $\mathfrak{C}(V)$ is precisely the set of all unions of subsets of V; thus $\mathfrak{C}(V)$ is exactly the image of $\mathbb{P}(V)$ under $\bigcap$. Moreover, $\mathfrak{D}(\mathfrak{C}(V))$ is the set of all unions of elements of $\mathfrak{C}(V)$. Therefore, for an element x to exist in $\mathfrak{D}(\mathfrak{C}(V))$, there must be a set S of elements of $\mathbb{P}(V)$, *i.e.*, there must be an $S \in \mathbb{P}(\mathbb{P}(V))$ such that $\mathfrak{F}(S) = x$. □

Lemma 8.47. *Let V be a finite set of sets. $\mathfrak{D}(\mathfrak{C}(V))$ is closed under union.*

Proof. Take two elements $a, b \in \mathfrak{D}(\mathfrak{C}(V))$. By Corollary 8.46, there exist $A, B \subset \mathbb{P}(V)$ such that $a = \mathfrak{F}(A)$ and $b = \mathfrak{F}(B)$

$$\begin{aligned}
a \cup b &= \mathfrak{F}(A) \cup \mathfrak{F}(B) \\
&= \bigcup\{\cap s \mid s \in A\} \cup \bigcup\{\cap s \mid s \in B\} \\
&= \bigcup\{\cap s \mid s \in (A \cup B)\} \\
&= \mathfrak{F}(A \cup B)
\end{aligned}$$

Since each element of $A \cup B$ is a subset of V, $a \cup b = \mathfrak{F}(A \cup B) \in \mathfrak{D}(\mathfrak{C}(V))$ by Corollary 8.45. □

Lemma 8.48. *Let V be a finite set of sets. $\mathfrak{D}(\mathfrak{C}(V))$ is closed under intersection.*

Proof. For this case, we use the Boolean identity

$$(X \cup Y) \cap (H \cup G) = (X \cap H) \cup (Y \cap H) \cup (Y \cap H) \cup (Y \cap G)\,.$$

Take $a, b \in \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$. By Corollary 8.46, there exist $A, B \subset \mathbb{P}(V)$ such that $a = \mathfrak{F}(A)$ and $b = \mathfrak{F}(B)$. $A, B \subset \mathbb{P}(V \cup \widetilde{V})$ means that A and B are of the form

$$\begin{aligned} A &= \{A_1, A_2, ..., A_{|A|}\} && \text{with } A_i \subset V \text{ for each } 1 \leqslant i \leqslant |A| \\ B &= \{B_1, B_2, ..., B_{|B|}\} && \text{with } B_i \subset V \text{ for each } 1 \leqslant i \leqslant |B|\,, \end{aligned}$$

where

$$\begin{aligned} A_i &= \{A_{i,1}, A_{i,2}, ..., A_{i,|A_i|}\} && A_{i,j} \in V \text{ for each } 1 \leqslant j \leqslant |A_i| \\ B_i &= \{B_{i,1}, B_{i,2}, ..., B_{i,|B_i|}\} && B_{i,j} \in V \text{ for each } 1 \leqslant j \leqslant |B_i|\,. \end{aligned}$$

Define the set $S = \{S_{i,j}\}_{\substack{1 \leqslant i \leqslant |A| \\ 1 \leqslant j \leqslant |B|}}$ where

$$S_{i,j} = \{A_{i,k} \mid 1 \leqslant k \leqslant |A_i|\} \cup \{B_{j,k} \mid 1 \leqslant k \leqslant |B_j|\}$$

Each $A_{i,k}$ and each $B_{j,k}$ is an element of V; so each $S_{i,j} \in \mathbb{P}(V)$ and $S \subset \mathbb{P}(V)$.
Now we can proceed to show $a \cap b \in \mathfrak{D}(\mathfrak{C}(V))$.

$$\begin{aligned} a \cap b &= \mathfrak{F}(A) \cap \mathfrak{F}(B) \\ &= \bigcup\{\cap s \mid s \in A\} \cap \bigcup\{\cap s \mid s \in B\} \\ &= (\bigcap A_1 \cup \bigcap A_2 \cup ... \cup \bigcap A_{|A|}) \cap (\bigcap B_1 \cup \bigcap B_2 \cup ... \cup \bigcap B_{|B|}) \\ &= (\bigcap A_1 \cap \bigcap B_1) \cup (\bigcap A_1 \cap \bigcap B_2) \cup ... \cup (\bigcap A_1 \cap \bigcap B_{|B|}) \\ &\quad \cup (\bigcap A_2 \cap \bigcap B_1) \cup (\bigcap A_2 \cap \bigcap B_2) \cup ... \cup (\bigcap A_2 \cap \bigcap B_{|B|}) \\ &\quad \cup ... \\ &\quad \cup (\bigcap A_{|A|} \cap \bigcap B_1) \cup (\bigcap A_{|A|} \cap \bigcap B_2) \cup ... \cup (\bigcap A_{|A|} \cap \bigcap B_{|B|}) \\ &= \bigcup_{\substack{1 \leqslant i \leqslant |A| \\ 1 \leqslant j \leqslant |B|}} (\bigcap A_i \cap \bigcap B_j) \\ &= \bigcup_{\substack{1 \leqslant i \leqslant |A| \\ 1 \leqslant j \leqslant |B|}} (\bigcap\{A_{i,1}, A_{i,2}, ..., A_{i,|A_i|}\} \cap \bigcap\{B_{j,1}, B_{j,2}, ..., B_{j,|B_j|}\}) \\ &= \bigcup_{\substack{1 \leqslant i \leqslant |A| \\ 1 \leqslant j \leqslant |B|}} \big((A_{i,1} \cap A_{i,2} \cap ... \cap A_{i,|A_i|}) \cap (B_{j,1} \cap B_{j,2} \cap ... \cap B_{j,|B_j|})\big) \\ &= \bigcup_{\substack{1 \leqslant i \leqslant |A| \\ 1 \leqslant j \leqslant |B|}} \bigcap S_{i,j} \\ &= \bigcup\{\cap s \mid s \in S\} \\ &= \mathfrak{F}(S) \end{aligned}$$

Since $S \subset \mathbb{P}(V)$ and $a \cap b = \mathfrak{F}(S)$, then by Corollary 8.45 $a \cap b \in \mathfrak{D}(\mathfrak{C}(V)$. □

Lemma 8.49. *Let V be a finite set of sets. $\mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$ is closed under relative complement.*

Proof. For this case, we use the Boolean identity

$$(X \cup Y) \backslash (H \cup G) = (X \backslash H) \cup (Y \backslash H) \cup (Y \backslash H) \cup (Y \backslash G)\,.$$

Since $a, b \in \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$, by Corollary 8.46, there exist $A \subset \mathbb{P}(V \cup \widetilde{V})$ and $B \subset \mathbb{P}(V \cup \widetilde{V})$ such that $a = \mathfrak{F}(A)$ and $b = \mathfrak{F}(B)$. $A, B \subset \mathbb{P}(V \cup \widetilde{V})$ means that A and B are of the form

$$\begin{aligned} A &= \{A_1, A_2, ..., A_{|A|}\} && \text{with } A_i \subset V \cup \widetilde{V} \text{ for each } 1 \leqslant i \leqslant |A| \\ B &= \{B_1, B_2, ..., B_{|B|}\} && \text{with } B_i \subset V \cup \widetilde{V} \text{ for each } 1 \leqslant i \leqslant |B| \,, \end{aligned}$$

where

$$\begin{aligned} A_i &= \{A_{i,1}, A_{i,2}, ..., A_{i,|A_i|}\} && A_{i,j} \in V \cup \widetilde{V} \text{ for each } 1 \leqslant j \leqslant |A_i| \\ B_i &= \{B_{i,1}, B_{i,2}, ..., B_{i,|B_i|}\} && B_{i,j} \in V \cup \widetilde{V} \text{ for each } 1 \leqslant j \leqslant |B_i| \,. \end{aligned}$$

Now we show that $a \backslash b \in \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$.

$$\begin{aligned} a \backslash b &= \mathfrak{F}(A) \setminus \mathfrak{F}(B) \\ &= \bigcup \{\cap s \mid s \in A\} \backslash \bigcup \{\cap s \mid s \in B\} \\ &= \big(\bigcap A_1 \cup \bigcap A_2 \cup ... \cup \bigcap A_{|A|}\big) \quad \setminus \quad \big(\bigcap B_1 \cup \bigcap B_2 \cup ... \cup \bigcap B_{|B|}\big) \\ &= \big(\bigcap A_1 \backslash \bigcap B_1\big) \cup \big(\bigcap A_1 \backslash \bigcap B_2\big) \cup ... \cup \big(\bigcap A_1 \backslash \bigcap B_{|B|}\big) \\ &\quad \cup \big(\bigcap A_2 \backslash \bigcap B_1\big) \cup \big(\bigcap A_2 \backslash \bigcap B_2\big) \cup ... \cup \big(\bigcap A_2 \backslash \bigcap B_{|B|}\big) \\ &\quad \cup ... \\ &\quad \cup \big(\bigcap A_{|A|} \backslash \bigcap B_1\big) \cup \big(\bigcap A_{|A|} \backslash \bigcap B_2\big) \cup ... \cup \big(\bigcap A_{|A|} \backslash \bigcap B_{|B|}\big) \\ &= \bigcup_{\substack{1 \leqslant i \leqslant |A| \\ 1 \leqslant j \leqslant |B|}} \big(\bigcap A_i \backslash \bigcap B_j\big) \\ &= \bigcup_{\substack{1 \leqslant i \leqslant |A| \\ 1 \leqslant j \leqslant |B|}} \big(\bigcap \{A_{i,1}, A_{i,2}, ..., A_{i,|A_i|}\} \setminus \bigcap \{B_{j,1}, B_{j,2}, ..., B_{j,|B_j|}\}\big) \\ &= \bigcup_{\substack{1 \leqslant i \leqslant |A| \\ 1 \leqslant j \leqslant |B|}} \big((A_{i,1} \cap A_{i,2} \cap ... \cap A_{i,|A_i|}) \setminus (B_{j,1} \cap B_{j,2} \cap ... \cap B_{j,|B_j|})\big) \\ &= \bigcup_{\substack{1 \leqslant i \leqslant |A| \\ 1 \leqslant j \leqslant |B|}} \big((A_{i,1} \cap A_{i,2} \cap ... \cap A_{i,|A_i|}) \cap (\overline{B_{j,1}} \cup \overline{B_{j,2}} \cup ... \cup \overline{B_{j,|B_j|}})\big) \\ &= \bigcup_{\substack{1 \leqslant i \leqslant |A| \\ 1 \leqslant j \leqslant |B|}} \bigcap_{\substack{1 \leqslant p \leqslant |A_i| \\ 1 \leqslant q \leqslant |B_j|}} (A_{i,p} \cup \overline{B_{j,q}}) \end{aligned}$$

We know $V \cup \widetilde{V} \subset \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V})$ because every element of $V \cup \widetilde{V}$ is a trivial union of intersection of itself. *E.g.*, if $\alpha \in V \cup \widetilde{V}$, then $(\alpha \cap \alpha) \cup (\alpha \cap \alpha) \in \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V})$.

Also, $A_{i,p}, \overline{B_{j,q}} \in V \cup \widetilde{V} \subset \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$. Since by Lemma 8.47, $\mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$ is closed under union, we know there exists a $X_{i,j,p,q} \in \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$ such that $X_{i,j,p,q} = A_{i,p} \cup \overline{B_{j,q}}$. So

$$a \backslash b = \bigcup_{\substack{1 \leqslant i \leqslant |A| \\ 1 \leqslant j \leqslant |B|}} \bigcap_{\substack{1 \leqslant p \leqslant |A_i| \\ 1 \leqslant q \leqslant |B_j|}} X_{i,j,p,q} \,.$$

Similarly, for each i, j, we know that $\bigcap_{\substack{1 \leqslant p \leqslant |A_i| \\ 1 \leqslant q \leqslant |B_j|}} X_{i,j,p,q}$, because by Lemma 8.48, $\mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$ is closed under intersection. Thus, there exists a $Y_{i,j} \in \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$ such that $Y_{i,j} = \bigcap_{\substack{1 \leqslant p \leqslant |A_i| \\ 1 \leqslant q \leqslant |B_j|}} X_{i,j,p,q}$. So

$$a \backslash b = \bigcup_{\substack{1 \leqslant i \leqslant |A| \\ 1 \leqslant j \leqslant |B|}} Y_{i,j} \,.$$

Finally, since $\mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$ is closed under union, $a \backslash b \in \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$ □

The following Lemma 8.50 claims that every element of $\mathscr{B}(V)$ can be expressed as a union of intersections of elements from V or their complements.

Lemma 8.50. *Let $V = \{D_1, D_2, ..., D_n\}$ be a finite set of sets, then $\mathscr{B}(V) \subset \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$.*

Proof. **By Structural induction:** Let $x \in \mathscr{B}(V)$. We wish to show $x \in \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$.

Case 1: $x \in V$: $x = \mathfrak{F}(\{\{x\}\}) \in \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$.

Case 2: $x \in \mathscr{B}(V) \backslash V$: By the inductive assumption, we claim $a, b \in \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$, such that $x \in \{a \cap b, a \cup b, a \backslash b\}$. We need to show $x \in \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$.

If $x = a \cup b$, then by Lemma 8.47, $\mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$ is closed under union; so $a \cup b \in \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$.

If $x = a \cap b$, then by Lemma 8.48, $\mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$ is closed under intersection; so $a \cap b \in \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$.

If $x = a \backslash b$, then by Lemma 8.49, $\mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$ is closed under relative complement; so $a \backslash b \in \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$.

□

Finally, the grand result of this section is that $\mathscr{B}(V)$ is a finite set.

Theorem 8.51. *If V is a finite set of sets, then $\mathscr{B}(V)$ has finite cardinality.*

Proof. Let $V = \{D_1, D_2, ..., D_n\}$ be a finite set of sets, and let $\widetilde{V} = \{\overline{D_1}, \overline{D_2}, ..., \overline{D_n}\}$. $\mathscr{B}(V) \subset \mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))$ by Lemma 8.50. Since $|V \cup \widetilde{V}| \leqslant 2n$, we know $|\mathfrak{C}(V \cup \widetilde{V})| \leqslant 2^{2n}$ by Lemma 8.41, and $|\mathfrak{D}(\mathfrak{C}(V \cup \widetilde{V}))| \leqslant 2^{2^{2n}}$ by Lemma 8.42. Finally $|\mathscr{B}(V)| \leqslant 2^{2^{2 \cdot |V|}}$. □

8.2.5 Disjoint decomposition

In this section, we define the notion of *disjoint decomposition* along with providing some examples. We also present and prove two results; Lemma 8.56, which is used to prove Theorem 8.65, and Lemma 8.61, which is used to prove Lemma 8.56. Lemma 8.56 in particular sheds light on how one might create an algorithm to improve or refine a given disjoint decomposition.

Definition 8.52. Given a set of non-empty, possibly overlapping sets $V \subseteq \mathbb{P}(U)$, set D is said to be a disjoint decomposition of V, if D is disjoined,$D \subseteq \mathscr{B}(V)$, and D exactly covers V.

Another way of thinking about Definition 8.52 is that if D is a disjoint decomposition of V, then D is a partition of $\bigcup V$, and that every element of D is a Boolean combination of elements of V. We will see in Theorem 8.65 that each of these Boolean combinations is an intersection with some element of V.

Example 8.53 (A disjoint decomposition)**.** Let $V = \{\{1, 2\}, \{2, 3\}\}$ as in Example 8.28. $D = \{\{1\}, \{2, 3\}\} \subseteq \mathscr{B}(V)$ is a disjoint decomposition because $\{1\} \parallel \{2, 3\}$, and $\bigcup D = \{1\} \cup \{2, 3\} = \{1, 2\} \cup \{2, 3\} = \bigcup V$.

Corollary 8.54. *If $V \subseteq \mathbb{P}(U)$ and $|V| < \infty$, a disjoint decomposition of V has finite cardinality.*

Proof. A disjoint decomposition of V is a subset of $\mathscr{B}(V)$, which has finite cardinality by Theorem 8.51. □

Corollary 8.55. *If $V \subseteq \mathbb{P}(U)$ with $|V| < \infty$, then there are only finitely many disjoint decompositions.*

Proof. Each disjoint decomposition is a subset of $\mathscr{B}(V)$ which is finite by Theorem 8.51 in Appendix 8.2.4. This means its power set, $\mathbb{P}(V)$ is finite. Thus there are only finitely many disjoint decompositions of V. □

Lemma 8.56. *Suppose D is disjoint decomposition of V, and suppose $X \in D$. If there exist distinct $\alpha, \beta \in \mathscr{B}(V)$, both different from $\varnothing$ such that $\{\alpha, \beta\}$ is a disjoint decomposition of $\{X\}$, then $\{\alpha, \beta\} \cup D\backslash\{X\}$ is a disjoint decomposition of V with cardinality $|D| + 1$.*

Proof. First we show that both α and β are disjoint from all elements of $D\backslash X$. Proof by contradiction. Without loss of generality take non-empty $\gamma \in D\backslash X$ such that $\alpha \nparallel \gamma$. So $\varnothing \neq \alpha \cap \gamma \subseteq D\backslash X$. So $\alpha \cap \gamma \not\subset X$. But since $\alpha \subseteq X$ we also have $\alpha \cap \gamma \subseteq X$. Contradiction!

Since $\alpha \cup \beta = X$, $\bigcup D = \alpha \cup \beta \cup \bigcup D\backslash\{X\}$. Thus $\{\alpha, \beta\} \cup D\backslash\{X\}$ is a cover of V.

Now since α and β are disjoint from all elements of $D\backslash\{X\}$ and disjoint from each other, we know $\alpha \notin D\backslash X$ and $\beta \notin D\backslash\{X\}$, $|\{\alpha, \beta\} \cup D\backslash\{X\}| = |\{\alpha, \beta\}| + |D\backslash\{X\}| = 2 + |D| - 1 = |D| + 1$ □

A brief intuitive explanation of Lemma 8.56 may be useful. The lemma basically says that if we start with a disjoint decomposition D and one of the elements, X, of that disjoint decomposition can itself be decomposed into $\{\alpha, \beta\} \subseteq \mathscr{B}(V)$, then we can construct a *better* disjoint decomposition having exactly one additional element, simply be removing X and adding back α and β. *I.e.* starting with D, if X is an element of D, and X is decomposable into $\{\alpha, \beta\}$, then a *better* disjoint decomposition is $\{\alpha, \beta\} \cup D\backslash\{X\}$.

Example 8.57 (Using Lemma 8.56 to improve a disjoint decomposition)**.**

Let $V = \{\{1, 2\}, \{2, 3\}\}$ as in Example 8.28.
Recall that $\mathscr{B}(V) = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.

$D = \{\{1\}, \{2, 3\}\}$ is a disjoint decomposition of V. Let $X = \{2, 3\}$, $\alpha = \{2\}$, $\beta = \{3\}$. Notice that $\alpha = \{2\} \in \mathscr{B}(V)$ and $\beta = \{3\} \in \mathscr{B}(V)$, that $\{2\} \parallel \{3\}$, and that $X = \{2, 3\} = \{2\} \cup \{3\} = \alpha \cup \beta$.

According to Lemma 8.56, $\{\alpha, \beta\} \cup D\backslash\{X\}$ is a disjoint decomposition of V with cardinality $|D| + 1$. Is this true? Yes!

$$\begin{aligned} D' &= \{\alpha, \beta\} \cup D\backslash\{X\} \\ &= \{\{2\}, \{3\}\} \cup \{\{1\}, \{2, 3\}\}\backslash\{\{2, 3\}\} \\ &= \{\{1\}, \{2\}, \{3\}\}. \end{aligned}$$

Moreover, $|D| = 2$, whereas $|D'| = 3$.

Lemma 8.58. *If $A \subseteq U$ and $B \subseteq U$, with $|A| = |B| < \infty$ but $A \neq B$, then $A\backslash B \neq \varnothing$ and $B\backslash A \neq \varnothing$.*

Proof. Proof by contradiction: If $A\backslash B = B\backslash A = \varnothing$ then $A \subseteq B$ and $B \subseteq A$; so $A = B$. Contradiction! So without loss of generality assume $A\backslash B = \varnothing$, and $A\backslash B \neq \varnothing$. We have $|A\backslash B| = 0$, and $|B\backslash A| > 0$.

$A = A\backslash B \cup A \cap B$
$A\backslash B \parallel A \cap B$, so $|A| = |A\backslash B \cup A \cap B| = |A\backslash B| + |A \cap B| = 0 + |A \cap B| = |A \cap B|$.
$B = B\backslash A \cup A \cap B$.
$B\backslash A \parallel A \cap B$, so $|B| = |B\backslash A \cup A \cap B| = |B\backslash A| + |A \cap B| > |A \cap B|$
$|A| = |B| > |A \cap B| = |A|$. Contradiction! □

Example 8.59 (Example of Lemma 8.58)**.** Let $A = \{1, 2, 3\}$, $B = \{1, 2, 4\}$. A and B fulfill Lemma 8.58. In particular $|A| = |B| = 3 < \infty$, and $A \neq B$. The lemma claims that $A\backslash B \neq \varnothing$ and $B\backslash A \neq \varnothing$, and this is in fact the case. $A\backslash B = \{1, 2, 3\}\backslash\{1, 2, 4\} = \{3\} \neq \varnothing$. $B\backslash A = \{1, 2, 4\}\backslash\{1, 2, 3\} = \{4\} \neq \varnothing$.

Notation 8.60. We use the symbol $\bigsqcup$ to indicate union of mutually disjoint sets. Moreover, if we write $\bigsqcup X_n$ then we claim or emphasize that $i \neq j \implies X_i \parallel X_j$.

Lemma 8.61. *If D_1 and D_2 are disjoint decompositions of V such that $D_1 \neq D_2$ and $|D_1| = |D_2|$, then there exists a disjoint decomposition of V with higher cardinality.*

Proof. Let D_1 and D_2 be disjoint decompositions of V with equal cardinality, and let $m = |D_1| = |D_2|$. Denote $D_1 = \{X_1, X_2, ..., X_m\}$ and $D_2 = \{Y_1, Y_2, ..., Y_m\}$. By Lemma 8.58 we can take $Y_j \in D_2$ such that $Y_j \notin D_1$, *i.e.* $Y_j \in D_2 \backslash D_1$. Since $Y_j \subseteq \bigsqcup D_2 = \bigsqcup D_1$, take $X_i \in D_1$ such that $Y_j \not\parallel X_i$. $Y_j \not\parallel X_i \implies Y_j \cap X_i \neq \varnothing$.

Case 1: $X_i \not\subseteq Y_j \implies X_i \backslash Y_J \neq \varnothing$. $X_i = X_i \backslash Y_j \cup X_i \cap Y_j$, with $X_i \backslash Y_j \parallel X_i \cap Y_j$, so by Lemma 8.56, $\{X_i \backslash Y_j, X_i \cap Y_j\} \cup D_1 \backslash \{X_i\}$ is disjoined and has cardinality $m + 1$.

Case 2: $X_i = Y_j$. Impossible since $X_i \in D_1$ while $Y_j \in D_2 \backslash D_1$.

Case 3: $X_i \subsetneq Y_j \implies Y_j \backslash X_i \neq \varnothing$. $Y_j = Y_j \backslash X_i \cup X_i \cap Y_j$, with $Y_j \backslash X_i \parallel X_i \cap Y_j$, so by Lemma 8.56, $\{Y_j \backslash X_i, X_i \cap Y_j\} \cup D_2 \backslash \{Y_j\}$ is disjoined and has cardinality $m + 1$.

□

Example 8.62 (Using Lemma 8.61 to improve a disjoint decomposition)**.**

Let $V = \{\{1, 2\}, \{2, 3\}\}$ as in Example 8.28. Let $D_1 = \{\{1\}, \{2, 3\}\}$ and $D_2 = \{\{1, 2\}, \{3\}\}$. Notice that both D_1 and D_2 are disjoint decompositions of V and that they fulfill the assumptions of Lemma 8.61. In particular, $D_1 \neq D_2$ and $|D_1| = |D_2| = 2$.

The lemma claims that there is therefore a disjoint decomposition with cardinality 3. Moreover, the proof of Lemma 8.61 suggests a way to find such a disjoint decomposition. To do this we must take $X \in D_1$ and $Y \in D_2$ such that $X \neq Y$, and $X \not\parallel Y$.

$$\begin{aligned} \text{Let's take} \quad & X = \{2, 3\} \in D_1 \\ \text{and} \quad & Y = \{1, 2\} \in D_2. \end{aligned}$$

We can now construct two sets D_1' and D_2', and one of these or the other (or perhaps both) will be a disjoint decomposition of cardinality 3.

$$\begin{aligned} D_1' &= \{X \backslash Y, X \cap Y\} \cup D_1 \backslash \{X\} \\ &= \{\{2, 3\} \backslash \{1, 2\}, \{2, 3\} \cap \{1, 2\}\} \cup \{\{1\}, \{2, 3\}\} \backslash \{\{2, 3\}\} \\ &= \{\{3\}, \{2\}\} \cup \{\{1\}\} \\ &= \{\{1\}, \{2\}, \{3\}\} \end{aligned}$$

$$\begin{aligned} D_2' &= \{Y \backslash X, X \cap Y\} \cup D_2 \backslash \{Y\} \\ &= \{\{1, 2\} \backslash \{2, 3\}, \{2, 3\} \cap \{1, 2\}\} \cup \{\{1, 2\}, \{3\}\} \backslash \{\{1, 2\}\} \\ &= \{\{1\}, \{2\}\} \cup \{\{3\}\} \\ &= \{\{1\}, \{2\}, \{3\}\} \end{aligned}$$

We see, in this case, that both D_1' and D_2' are disjoint decompositions of cardinality 3.

It happens in this example that $D_1' = D_2'$. This equality is, however, not an immediate consequence of Lemma 8.61.

8.2.6 Maximal disjoint decomposition

Definition 8.63. If D is a disjoint decomposition such that for any other disjoint decomposition D' it holds that $|D| \geqslant |D'|$, then D is said to be a maximal disjoint decomposition.

We use $|D| \geqslant |D'|$ in Definition 8.63, rather than $|D| > |D'|$, simply because we have not yet made a uniqueness argument. For the moment we allow the possibility that two or more such sets exist. If D has the property that any other decomposition is the same size or smaller, then D is a maximal disjoint decomposition. We will argue for uniqueness in Theorem 8.67.

Even though we have not yet argued for the uniqueness of such a decomposition, the intuition here is that the *maximal disjoint decomposition* of a given set V of subsets of U is the most thorough partitioning of $\bigcup V$ which is possible when we are only allowed to use Boolean combinations of the sets given in V. *I.e.*, while it might be possible to conceive of a better partitioning (more complete) and in fact, it may not even be possible in general to find the most thorough partitioning in general, when restricted to using only Boolean combinations of elements of V, we can think of the most *thorough* such partitioning.

We avoid using the term *largest decomposition* to avoid a potential confusion. By *most thorough*, we mean the set, $D = \{X_1, X_2, ..., X_m\}$ of subsets for which D has the maximum possible number of elements (maximizing m), not such that the X's themselves has as many elements as possible. I.e., we want a large collection of small sets, rather than a small collection of large sets.

Example 8.64 (a maximal disjoint decomposition of a set)**.** Let

$$V = \{\{1,2\},\{2,3,4\}\}\,;$$

and

$$\mathscr{B}(V) = \{\varnothing,\{1\},\{2\},\{1,2\},\{3,4\},\{1,3,4\},\{2,3,4\},\{1,2,3,4\}\}\,.$$

Then

$$D = \{\varnothing,\{1\},\{2\},\{3,4\}\}$$

is the maximal disjoint decomposition of V.

Why? Clearly D is a disjoint decomposition of V. The only way to decompose further would be to consider the set $D' = \{\varnothing,\{1\},\{2\},\{3\},\{4\}\}$. However, D' is not the maximal disjoint decomposition of V because $D' \not\subset \mathscr{B}(V)$. Notice that $\{3\} \notin \mathscr{B}(V)$ (and neither is $\{4\}$). There is no way to produce the set $\{3\}$ (nor $\{4\}$) starting with the elements of V and combining them finitely many times using intersection, union, and relative complement.

Theorem 8.65. *If D is a maximal disjoint decomposition of V, then $\forall X \in D \;\; \exists\, A \in V$ such that $X \subseteq A$.*

Proof. Proof by contradiction: Let D be a maximal disjoint decomposition of V. Since $\bigcup D = \bigcup V$, let $Y \in V$ such that $X \nparallel Y$. By contrary assumption $X \not\subset Y \implies X\backslash Y \neq \varnothing$. So $X = X\backslash Y \cup X \cap Y$, $X\backslash Y \parallel X \cap Y$, so by Lemma 8.56, $\{X\backslash Y, X \cap Y\} \cup \bigcup D\backslash X$ is disjoint decomposition of V with greater cardinality than D, which is impossible since D was assumed to be a maximal disjoint decomposition. Contradiction! □

Theorem 8.65 basically says that every element of a maximal disjoint decomposition of V is a subset of some element of V. This means that if we want to construct a maximal disjoint decomposition algorithmically, an interesting strategy might be to start with the elements of V and look at cleverly constructed intersections thereof. This is in fact the strategy we will apply in the algorithms explained in Section 9.1 and 9.4.

Example 8.66 (Maximal disjoint decomposition)**.** As in Example 8.64, let $V = \{\{1,2\},\{2,3,4\}\}$. $D = \{\varnothing,\{1\},\{2\},\{3,4\}\}$ is the maximal disjoint decomposition of V. For each $X \in D$ we can find an $A \in V$

such that $X \subseteq A$. Note, that A is not necessarily unique. In particular.

$$D \; s.t. \; \varnothing \subseteq \{1,2\} \in V$$
$$D \; s.t. \; \{1\} \subseteq \{1,2\} \in V$$
$$D \; s.t. \; \{2\} \subseteq \{2,3,4\} \in V$$
$$D \; s.t. \; \{3,4\} \subseteq \{2,3,4\} \in V$$

Theorem 8.67. *There exists a unique maximal disjoint decomposition.*

Proof. Let $\mathscr{D}$ be the set of all disjoint decompositions of V. $\mathscr{D}$ is a finite set by Corollary 8.55. Let C be the set of cardinalities of elements of $\mathscr{D}$, each of which is finite by Corollary 8.54. C is a finite set of integers, let $M = max(C)$. Let D_{max} be the set of elements $X \in \mathscr{D}$ such that $|X| = M$. We now show that $|D_{max}| = 1$.

Case $|D_{max}| = 0$: Impossible because there exists at least one decomposition, namely the trivial one: $\{\bigcup V\}$.

Case $|D_{max}| >= 2$: Impossible because if $\alpha, \beta \in D_{max}, \alpha \neq \beta$, then by Lemma 8.61 there exists another disjoint decomposition, γ such that $|\gamma| > |\alpha|$, which would mean that α is not maximal.

Case $|D_{max}| = 1$: Since $|D_{max}|$ cannot be negative, 0, nor greater than 1, it must be equal to 1.

Thus there is exactly one disjoint decomposition whose cardinality is larger than any other disjoint decomposition. □

8.3 Related work

Newton *et al.* [NDV16, NV18b] presented this problem when attempting to determinize automata used to recognize rational type expressions. It was stated in that work that the algorithm employed there had performance issues which needed to be addressed. Newton *et al.* showed such performance improvements in a follow-up paper [NVC17].

We borrow the notation $\mathscr{B}(V)$ from measure theory [Str81, DS88], where it normally denotes Borel sets which are particular sigma algebras. The sigma algebra of V is often denoted as $\sigma(V)$ in measure theory literature. However, in our notation, we reserve the notation $\sigma(V)$ to denote the standard deviation. Hence, we invent the non-standard notation $\mathscr{B}(V)$ to denote sigma algebra.

The Baker `subtypep` procedure [Bak92, Val18] seeks to find distinguishing representatives for the non-numerical base types in the Common Lisp implementation. Baker proposes to order characteristic elements so that the same indexes serve as bit positions in bit vectors. The bit vectors are then combined with bit-wise Boolean operations in a manner similar to the set-wise Boolean operations shown in Figure 8.3. Baker argues that these bit-wise operations lead to more efficient execution than the run-time logic necessary for answer subset predicates using set-wise Boolean arithmetic.

Chapter 9

Calculating the MDTD

In Chapter 4 we introduced the problem of efficiently recognizing a sequence of object in Common Lisp given a regular type expression. This problem lead to two challenges called the MDTD problem and the serialization problem. In Chapter 11 we will look at the serialization problem. In the current chapter, we look at algorithms for the finding solution of the MDTD problem, which we showed, in Chapter 8, to uniquely exist. We base some of the strategies on s-expression based computation as presented in Chapter 2, and other strategies on the ROBDD generalizations which were presented in Chapter 7. We will analyze the performance of the different variations of the algorithms in Chapter 10.

Aside from presenting several MDTD algorithms, the current chapter uses the MDTD problem as a vehicle for comparing performance characteristics of the two data structure approaches: *i.e.* s-expression based type specifiers *vs.* ROBDDs.

We remind the reader of the MDTD problem. In the Venn diagram in Figure 8.1, $V = \{A_1, A_2, ..., A_8\}$. We wish to construct logical combinations of the sets A_1, A_2, ..., A_8, to form as many mutually disjoint subsets as possible. We know, from Corollary 8.54, that it is not possible to construct infinitely many such Boolean combinations. The resulting *decomposition* should have the same union as the original set. The maximal disjoint decomposition $D = \{X_1, X_2, ..., X_{13}\}$ of V is shown in Figure 8.3.

Definition 9.1. Let U be a set and V be a set of subsets of U. The Sigma Algebra of V, denoted $\mathscr{B}(V)$, is the (smallest) superset of V such that

$$\alpha, \beta \in \mathscr{B}(V) \implies \{\alpha \cap \beta, \alpha \cap \overline{\beta}\} \subset \mathscr{B}(V)\,.$$

Definition 9.2 (Maximal disjoint decomposition)**.** Let U be a set, and let V and D be finite sets of non-empty subsets of U. D is said to be a disjoint decomposition of V if the elements of D are mutually disjoint, $D \subset \mathscr{B}(V)$, and $\bigcup_{X \in D} X = \bigcup_{A \in V} A$. If no larger set fulfills those properties then D is said to be the maximal disjoint decomposition of V.

Definition 9.3 (The MDTD problem)**.** Given a set U and a set of subsets thereof, let $V = \{A_1, A_2, ..., A_M\}$. Suppose that for each pair (A_i, A_j), we know which of the relations hold: $A_i \subset A_j$, $A_i \supset A_j$, $A_i \parallel A_j$. We would like to compute the maximal disjoint decomposition of V.

Implementing an algorithm to solve this problem is easy when we are permitted to look into the sets and partition the individual elements. Such access makes it easy to decide whether two given sets have an intersection and to calculate that intersection. In some programming languages, Common Lisp included, a type can be thought of as a set of (potential) values [Ans94, Section Type], Definition 2.1. In this case, set decomposition is really *type decomposition*. In general, we are not granted access to the individual elements, and we are not allowed to iterate over the elements, as the sets may be infinite. For the algorithm to work, we must have operators to test for set-equality, disjoint-ness and subset-ness (subtype-ness). It turns out that if

we have an empty type and binary subset predicate, then it is possible to express predicates for equality and disjointness in terms of them.

When attempting to implement an algorithm to solve the MDTD problem, the developer finds it necessary to choose a data structure to represent type specifiers. Which ever data structure is chosen to represent types, the program must calculate intersections, unions, relative complements, equivalence checks, and vacuity checks. As discussed in Section 2.5, s-expressions (*i.e.* lists and symbols) are a valid choice of data structure and the aforementioned operations may be implemented as list constructions and calls to the `subtypep` predicate.

As introduced in Chapter 5, another choice of data structure is the BDD. Using the BDD data structure along with the algorithms described in Chapter 5, we can efficiently represent and manipulate Common Lisp type specifiers. We may programmatically represent Common Lisp types largely independent of the actual type specifier representation. For example, the following two type specifiers denote the same set of values: `(or number (and array (not vector)))` and `(not (and (not number) (or (not array) vector)))`, and both are represented by the BDD as shown in Figure 9.1. Moreover, unions, intersections and relative complements of Common Lisp type specifiers can be calculated using the reduction BDD manipulation rules also explained in Chapter 5.

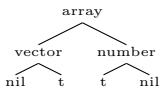


Figure 9.1: BDD representing `(or number (and array (not vector)))`

Newton *et al.* [NDV16] presented this problem along with a brute force solution, which we summarize here in Algorithm 9.1. Newton *et al.* noted that the algorithm had performance issues and more research was needed to improve it. In the current chapter we summarize our efforts to improve the algorithm. We tried several approaches which are as follows:

- The original, baseline, algorithm, shown in Section 9.1.
- Small optimizations in the baseline algorithm, explained in Section 9.2.
- Alternative SAT-like approach,explained Section 9.3.
- Alternative graph-based approach, explained in Section 9.4.
- Alternative data structure, the ROBDD, explained in Section 9.5.
- Alternative implementation of `subtypep`, explained in Section 9.6.

We implemented several algorithms using the approaches mentioned above and then compared the resulting computation times. Some results of the analysis can be seen in Chapter 10.

9.1 Baseline set disjoint decomposition

This section presents a conceptually simple disjoint set decomposition algorithm which we will refer to as the baseline algorithm. It was first presented at in [NDV16], where it was noted to have performance problems, and it was suggested that more research was needed. We address those performance concerns in the following sections.

The baseline algorithm is shown as pseudocode in Algorithm 6. This brute force algorithm is straightforward [DL15]. It heavily depends on the Common Lisp `subtype` and the Common Lisp functions shown in Figure 2.10. A compelling feature of this algorithm is that it easily fits in only 40 lines of Common Lisp code.

The code in Appendix B shows the actual implementation. The code follows that shown in Algorithm 6 but with a few optimizations added. In particular the $\|$ relation tested on lines 6.4 and 6.10 has been memoized. Additionally, lines 6.13 through 6.18 have been merged into a single operation.

A light examination of Algorithm 6 reveals that the computational complexity risks being $\mathcal{O}(n^3)$, where n is the size of the given set of sets. There are two sources of this complexity. Line 6.4 is an $\mathcal{O}(n^2)$ search for all the elements of U which are disjoint from all other elements of U. Line 6.10 is another $\mathcal{O}(n^2)$ search for a pair of sets which are not disjoint. And these two loops are repeated by the `while` loop on line 6.3. This complexity can be reduced.

There is another source of complexity which is less obvious. This hidden complexity lies in the fact that the variables, D and U, in the pseudocode (and corresponding variables in the Common Lisp code) represent sets. This means that when elements are added to the sets, uniqueness must be maintained, which is an $\mathcal{O}(n^2)$ or $\mathcal{O}(n \log(n))$ operation depending how it is implemented. This $\mathcal{O}(n^2)$ (or $\mathcal{O}(n \log(n))$) operation may be a significant computation depending on the comparison function, which may be a simple pointer comparison in some cases, or a tree search in other cases as will be seen.

Algorithm 6: Baseline function to find the maximal disjoint decomposition

Input: A finite non-empty set U of sets
Output: A finite set D of disjoint sets

```
6.1  begin
6.2      D ← ∅
6.3      while true do
6.4          D′ ← {u ∈ U | u′ ∈ U\{u} ⟹ u ∥ u′}
6.5          D ← D ∪ D′
6.6          U ← U\D′
6.7          if U = ∅ then
6.8              return D
6.9          else
6.10             Find X ∈ U and Y ∈ U such that X ∦ Y
6.11             if X = Y then
6.12                 U ← U\{Y}
6.13             else if X ⊂ Y then
6.14                 U ← U\{Y} ∪ {Y\X}
6.15             else if Y ⊂ X then
6.16                 U ← U\{X} ∪ {X\Y}
6.17             else
6.18                 U ← U\{X, Y} ∪ {X ∩ Y, X\Y, Y\X}
6.19     return D
```

Notes about Algorithm 6:

Line 6.4: We find the set D' of all elements of U which are disjoint from all other elements of U. Notice that in line 6.4, if U is a singleton set, then $D' \leftarrow U$, thus $U \leftarrow \varnothing$ on line 6.6.

Line 6.4: This is of course an $\mathcal{O}(n^2)$ search, and it is repeated each time through the loop headed on line 6.3; the search therefore has $\mathcal{O}(n^3)$ complexity. Part of the motivation for the algorithm in Section 9.4 is to eliminate this $\mathcal{O}(n^3)$ search.

Line 6.8: If $U = \varnothing$ then we have collected all the disjoint sets into D.

Line 6.10: This search is $\mathcal{O}(n^2)$.

Line 6.10: It is guaranteed that X and Y exist because $|U| > 1$, and if all the elements of U were mutually disjoint, then they would have all been collected in line 6.4.

Line 6.13: The case analysis here does the following: $U \leftarrow \{X \cap Y, X\backslash Y, Y\backslash X\} \cup U\backslash\{\varnothing\}$. However, some elements of $\{X \cap Y, X\backslash Y, Y\backslash X\}$ may be $\varnothing$ or in fact X or Y depending on the subset relation between X and Y, thus the three cases specialize the possible subset relations.

Line 6.14: If $X \subset Y$, then $X \cap Y = X$ and $X\backslash Y = \varnothing$. Thus update U by removing Y, and adding $Y\backslash X$.

Line 6.16: If $Y \subset X$, then $X \cap Y = Y$ and $Y\backslash X = \varnothing$. Thus update U by removing X, and adding $X\backslash Y$.

Line 6.18: Otherwise, update U by removing X and Y, and adding $X \cap Y$, $X\backslash Y$, and $Y\backslash X$.

9.2 Small improvements in baseline algorithm

Algorithm 7 is an attempted improvement over the baseline algorithm, eliminating the $\mathcal{O}(n^2)$ operation on line 6.4 by selecting an arbitrary element A on line 7.4. A is either disjoint from all other elements of U, or there is a non-empty set $\mathcal{I}$ of all the elements of U which intersect A. An $\mathcal{O}(n)$ iteration on lines 7.6 and 7.5 determines two sets $\mathcal{I}$ and D', either of which may be empty. If $\mathcal{I}$ is empty, the consequence on line 7.9 is simply to add A to the output set D. However, if $\mathcal{I}$ is non-empty, then the $\mathcal{O}(n)$ iteration on lines 7.11, 7.12, and 7.13 calculates certain subsets of A and each element of $\mathcal{I}$ to add to U.

Algorithm 7: Improved complexity of baseline algorithm

Input: A finite non-empty set U of sets
Output: A finite set D of disjoint sets

```
7.1  begin
7.2      D ← ∅
7.3      while U ≠ ∅ do
7.4          α ← any element of U
7.5          D′ ← {u ∈ U | u ∥ α, u ≠ α}
7.6          I ← (U\D′)\{α}
7.7          U ← D′
7.8          if I = ∅ then
7.9              D ← D ∪ {α}
7.10         else
7.11             for β ∈ I do
7.12                 C ← {α ∩ β, α\β, β\α} \ {∅}
7.13                 U ← U ∪ C
7.14     return D
```

Notes about Algorithm 7:

Line 7.4: Here we choose some element of U. If the data structure representing U is a `list`, then taking the first element is sufficient. This operation may appear simply to be a loop over the elements of U, but keep in mind that U is modified each time through the *while* loop.

Line 7.6: $\mathcal{I}$ and D' can be calculated as a single loop through U as a partition based on the $\parallel$ relation. This operation effectively partitions U into three sets, the singleton $\{\alpha\}$, the elements of U which intersect α, and the elements of U which are disjoint from α. *I.e.*, $U = \{\alpha\} \sqcup \mathcal{I} \sqcup D'$

Line 7.6: Note that, depending on the implementation language and the data structures used, the calculation needed to determine whether two sets intersect might involve actually calculating the intersection. If this is the case, it is a good idea to save these intersections because they will be useful later on line 7.13.

Line 7.7: Remove α from U, and remove from U everything that is non disjoint from α, leaving only the elements which are disjoint from α.

Line 7.9: Add α to D. It is possible that $\alpha \in D$ already, either in the same syntactic form or in some other equivalent form. Therefore, a membership check should be made to avoid duplicate elements in D. The implementation of this membership check depends on the data structure used to represent these sets, and thus this check may be expensive.

Line 7.12: The set $\{\alpha \cap \beta, \alpha\backslash\beta, \beta\backslash\alpha\}$ is the same as the set $\{\alpha \cap \beta, \alpha\backslash(\alpha \cap \beta), \beta\backslash(\alpha \cap \beta)\}$. Depending on the data structures used, one of these representations may be easier to compute than the other.

Line 7.12: The set $C = \{\alpha \cap \beta,\ \alpha\backslash\beta,\ \beta\backslash\alpha\} \setminus \{\varnothing\}$ may have cardinality precisely 1, 2, or 3. $|C| = 3$ only in the case that $\alpha \cap \beta$, $\alpha\backslash\beta$ and $\beta\backslash\alpha$ are all different and non-empty. However, it can (and often does) happen that $\alpha\backslash\beta = \varnothing$ (if $\alpha \subset \beta$), $\beta\backslash\alpha = \varnothing$ (if $\alpha \supseteq \beta$), or $\alpha\backslash\beta = \beta\backslash\alpha = \varnothing$ (if $\alpha = \beta$). In any of these cases $|C| < 3$. We are sure that $|C| > 0$, because by construction $\alpha \cap \beta \neq \varnothing$. Thus $|C| \in \{1, 2, 3\}$.

9.3 Type disjoint decomposition as SAT problem

This problem of how to decompose sets, like those shown in Figure 8.1, into disjoint subsets as shown in Figure 8.3 can be viewed as a variant of the well known *Satisfiability Problem*, commonly called SAT [HMU06]. The problem is this: given a Boolean expression in n variables, find an assignment (either *true* or *false*) for each variable which makes the expression evaluate to *true*. This problem is known to be NP-Complete.

The approach is to consider the correspondence between the solutions of the Boolean equation: $A_1 + A_2 + ... + A_M$, versus the set of subsets of $A_1 \cup A_2 \cup ... \cup A_M$. Just as we can enumerate the $2^M - 1 = 255$ solutions of $A_1 + A_2 + ... + A_M$, we can analogously enumerate the subsets of $A_1 \cup A_2 \cup ... \cup A_M$.

discard	0000 0000	$\overline{A_1} \cap \overline{A_2} \cap \overline{A_3} \cap \overline{A_4} \ \cap \ \overline{A_5} \cap \overline{A_6} \cap \overline{A_7} \cap \overline{A_8}$
1	1000 0000	$A_1 \cap \overline{A_2} \cap \overline{A_3} \cap \overline{A_4} \ \cap \ \overline{A_5} \cap \overline{A_6} \cap \overline{A_7} \cap \overline{A_8}$
2	0100 0000	$\overline{A_1} \cap A_2 \cap \overline{A_3} \cap \overline{A_4} \ \cap \ \overline{A_5} \cap \overline{A_6} \cap \overline{A_7} \cap \overline{A_8}$
3	1100 0000	$A_1 \cap A_2 \cap \overline{A_3} \cap \overline{A_4} \ \cap \ \overline{A_5} \cap \overline{A_6} \cap \overline{A_7} \cap \overline{A_8}$
	...	...
254	1111 1110	$A_1 \cap A_2 \cap A_3 \cap A_4 \ \cap A_5 \cap A_6 \cap A_7 \cap \overline{A_8}$
255	1111 1111	$A_1 \cap A_2 \cap A_3 \cap A_4 \ \cap A_5 \cap A_6 \cap A_7 \cap A_8$

Figure 9.2: Correspondence of Boolean true/false equation with Boolean set equation

If we assume $M = 8$ as in Figure 8.1, the approach here is to consider every possible solution of the Boolean equation: $A_1 + A_2 + ... + A_8$. There are $2^8 - 1 = 255$ such solutions, because every 8-tuple of 0's and 1's is a solution except 0000 0000. If we consider the enumerated set of solutions: 1000 0000, 0100 0000, 1100 0000, ... 1111 1110, 1111 1111. We can analogously enumerate the potential subsets of the union of the sets shown in Figure 8.3: $A_1 \cup A_2 \cup A_3 \cup A_4 \cup A_5 \cup A_6 \cup A_7 \cup A_8$. Each subset is a potential solution representing an intersection of sets in $\{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8\}$. Such a correspondence is shown in Figure 9.2.

The only thing that remains is to eliminate the intersections which can be proven to be empty. For example, we see in 8.1 that A_1 and A_7 are disjoint, which implies $\varnothing = A_1 \cap A_7$, which further implies that every line of Figure 9.2 which contains A_1 and A_7 is $\varnothing$. This eliminates 64 lines. For example line 255: $\varnothing = A_1 \cap A_2 \cap A_3 \cap A_4 \cap A_5 \cap A_6 \cap A_7 \cap A_8$, and 63 other such lines.

In this, as in all SAT problems, some of these 2^8 possibilities can be eliminated because of known constraints. The constraints are derived from the known subset and disjoint-ness relations of the given sets. Looking at the Figure 8.1 we see that $A_5 \subset A_8$, which means that $A_5 \cap \overline{A_8} = \varnothing$. So we know that all solutions, where $A_5 = 1$ and $A_8 = 0$, can be eliminated. This elimination by constraint $A_5 \cap \overline{A_8} = \varnothing$ and the previous one, $A_1 \cap A_7 = \varnothing$, are each represented in the Boolean equation as a multiplication (Boolean multiply) by $\overline{A_1 A_7}$ and $\overline{A_5 \overline{A_8}}$: $(A_1 + A_2 + A_3 + A_4 + A_5 + A_6 + A_7 + A_8) \cdot \overline{A_1 A_7} \cdot \overline{A_5 \overline{A_8}}$.

There are as many as $\frac{8 \times 7}{2} = 28$ possible constraints imposed by pair relations. For each $\{X, Y\} \subset \{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8\}$:

Subset If $X \subset Y$, multiply the by constraint $\overline{X \overline{Y}} = (\overline{X} + Y)$

Superset If $Y \subset X$, multiply by the constraint $\overline{\overline{X} Y} = (X + \overline{Y})$

Disjoint If $X \cap Y = \varnothing$, multiply by the constraint $\overline{XY} = (\overline{X} + \overline{Y})$.

Otherwise no constraint.

A SAT solver will normally find one solution. That's just how they traditionally work. But the SAT flow can easily be extended so that once a solution is found, a new constraint can be generated by logically negating that solution, allowing the SAT solver to find a second solution. For example, when it is found that 1111 0000 (corresponding to $A_1 \cap A_2 \cap A_3 \cap A_4 \cap \overline{A_5} \cap \overline{A_6} \cap \overline{A_7} \cap \overline{A_8}$) is a solution, the equation can thereafter be multiplied by the new constraint (A_1 A_2 A_3 A_4 $\overline{A_5}$ $\overline{A_6}$ $\overline{A_7}$ $\overline{A_8}$), allowing the SAT solver to find yet another solution if such exists.

The process continues until there are no more solutions.

As a more concrete example of how the SAT approach works when applied to Common Lisp types, consider the case of the three types `array`, `sequence`, and `vector`. Actually, `vector` is the intersection of `array` and `sequence`.

First the SAT solver constructs (explicitly or implicitly) the set of candidates corresponding to the Lisp types.

```
(and array       sequence       vector)
(and array       sequence       (not vector))
(and array       (not sequence) vector)
(and array       (not sequence) (not vector))
(and (not array) sequence       vector)
(and (not array) sequence       (not vector))
(and (not array) (not sequence) vector)
(and (not array) (not sequence) (not vector))
```

The void one `(and (not array) (not sequence) (not vector))` can be immediately disregarded.

Since `vector` is a subtype of `array`, all types which include both `(not array)` and also `vector` can be disregarded: `(and (not array) sequence vector)` and `(and (not array) (not sequence) vector)`. Furthermore, since `vector` is a subtype of `sequence`, all types which include both `(not sequence)` and also `vector` can be disregarded. `(and array (not sequence) vector)` and `(and (not array) (not sequence) vector)` (which has already been eliminated by the previous step). The remaining ones are:

```
(and array       sequence       vector)       = vector
(and array       sequence       (not vector)) = nil
(and array       (not sequence) (not vector)) = (and array (not vector))
(and (not array) sequence       (not vector)) = (and sequence (not vector))
```

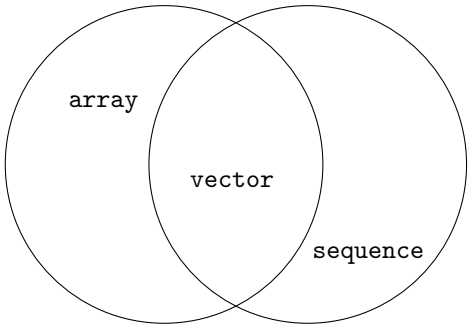


Figure 9.3: Relation of `vector`, `sequence`, and `array`. `vector` is identically the intersection of `array` and `sequence`.

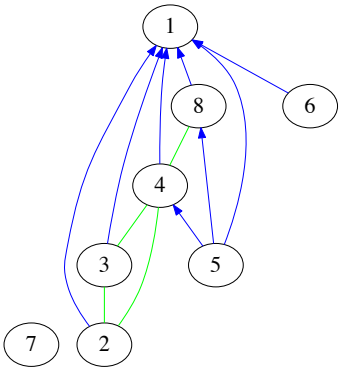
The algorithm returns a false positive. Unfortunately, this set still contains the `nil`, empty, type `(and array sequence (not vector))`. Figure 9.3 shows the relation of the Common Lisp types: `array`, `vector`, and `sequence`. We can see that `vector` is the intersection of `array` and `sequence`. The algorithm discussed above failed to introduce a constraint corresponding to this identity which implies that $array \cap sequence \cap \overline{vector} = \varnothing$.

It seems the SAT algorithm greatly reduces the search space, but is not able to give the minimal answer. The resulting types must still be tested for vacuity. This is easy to do, just use the `subtypep` function to test whether the type is a subtype of `nil`. E.g., `(subtypep '(and array sequence (not vector)) nil)` returns `t`. There are cases where the `subtypep` will not be able to determine the vacuity of a set. Consider the example: `(and fixnum (not (satisfies oddp)) (not (satisfies evenp)))`.

9.4 Set disjoint decomposition as graph problem

One of the sources of inefficiency of Algorithm 6 explained in Section 9.1 is that at each iteration of the loop, an $\mathcal{O}(n^2)$ search is made to find sets which are disjoint from all remaining sets. This search can be partially obviated if we employ a little extra book-keeping. The fact to realize is that if $X \parallel A$ and $X \parallel B$, then we know *a priori* that $X \parallel A \cap B$, $X \parallel A \backslash B$, $X \parallel B \backslash A$. This knowledge eliminates some of the useless operations.

In this section we present two similar variations of the algorithm, Algorithm 8 and Algorithm 9. An example run of the algorithm is detailed in Appendix F.



Node	Boolean expression
1	A_1
2	A_2
3	A_3
4	A_4
5	A_5
6	A_6
7	A_7
8	A_8

Figure 9.4: State 0: Topology graph

This algorithm is semantically similar to the baseline algorithm from Section 9.1, but rather than relying on Common Lisp primitives to make decisions about connectivity of sets/types, it initializes a graph representing the initial relationships, and thereafter manipulates the graph maintaining connectivity information. This algorithm is more complicated in terms of lines of code, 250 lines of Common Lisp code as opposed to 40 lines for the baseline algorithm.

This more complicated algorithm is presented here for two reasons. (1) It has much faster execution times, especially for larger sets types. (2) We hope that presenting the algorithm in a way which obviates the need to use Common Lisp primitives makes it evident how the algorithm might be implemented in a programming language other than Common Lisp.

Figure 9.4 shows a graph representing the topology (connectedness) of the Venn diagram shown in Figure 8.1. Nodes ①, ②, ... ⑧ in Figure 9.4 correspond respective to X_1, X_2, ... X_8 in the same figure.

The algorithm commences by constructing a graph from a given set of subsets (Section 9.4.1), and proceeds by breaking the green and blue connections, one by one, in controlled ways until all the nodes become isolated. Sometimes it is necessary to temporarily create new connections when certain connections are broken as is seen below. There are a small number of cases to consider as are explained in detail in Sections 9.4.2, 9.4.3, 9.4.4, and 9.4.5. Repeat alternatively applying both tests until all the nodes become isolated.

There are several possible flows for the algorithm. We start by two alternative basic flows in Algorithms 8 and 9. We discuss other variations of these flows in Section 10.5.

Algorithm 8: DecomposeByGraph-1

Input: U: A finite non-empty set of sets, *i.e.* U has no repeated elements and no empty elements.
Output: The maximal decomposition of U

```
8.1 begin
8.2     G ← ConstructGraph(U)
8.3     while G.blue or G.green do
8.4         for (X → Y) ∈ G.blue do
8.5             BreakRelaxedSubset(G, X, Y);
8.6         for {X, Y} ∈ G.green do
8.7             BreakTouching(G, X, Y);
8.8     return G.disjoint
```

Algorithm 9: DecomposeByGraph-2

Input: U: A finite non-empty set of sets, *i.e.* U has no repeated elements and no empty elements.
Output: The maximal decomposition of U

```
9.1  begin
9.2      G ← ConstructGraph(U)
9.3      while G.blue or G.green do
9.4          for (X → Y) ∈ G.blue do
9.5              BreakStrictSubset(G, X, Y);
9.6          for {X, Y} ∈ G.green do
9.7              BreakTouching(G, X, Y);
9.8          for (X → Y) ∈ G.blue do
9.9              BreakLoop(G, X, Y);
9.10     return G.disjoint
```

9.4.1 Graph construction

To construct this graph, first eliminate duplicate sets. *I.e.*, if $X \subset Y$ and $X \supseteq Y$, then discard either X or Y. It is necessary to consider each pair (X, Y) of sets, which gives an $\mathcal{O}(n^2)$ loop. Algorithm 10 describes the graph construction and uses functions defined in Algorithms 12, 13, 14, and 15.

Sometimes the relation of two sets cannot be determined. Such cases must be interpreted as follows:

- A blue arrow between two nodes means a subset relation.
- Neither blue arrow nor green line between two nodes means the disjoint relation.
- A green line between two nodes means the sets *may* touch.

Algorithm 12: AddGreenLine

Input: G: a graph
Input: X: a node of G
Input: Y: a node of G
Side Effects: Modifies G adding a green line between X and Y

```
12.1 begin
12.2     Push {X, Y} onto G.green
12.3     Push X onto Y.touches
12.4     Push Y onto X.touches
```

Algorithm 13: DeleteGreenLine

Input: G, a graph
Input: X: a node of G
Input: Y: a node of G
Side Effects: Modifies G deleting the green line between X and Y. Perhaps extends $G.disjoint$ and shortens $G.nodes$.

```
13.1 begin
13.2     Remove {X, Y} from G.green
13.3     Remove X from Y.touches
13.4     Remove Y from X.touches
13.5     MaybeDisjointNode(G, X)
13.6     MaybeDisjointNode(G, Y)
```

Algorithm 10: ConstructGraph

Input: A finite non-empty set U of sets, *i.e.* U has no repeated elements.

Output: A graph, G, in particular a set of blue ordered edges and green ordered edges. The nodes of nodes of G are some (or all) of the elements of U

```
10.1  begin
10.2      G.blue ← ∅
10.3      G.green ← ∅
10.4      G.nodes ← set of labeled nodes, seeded from U
10.5      G.disjoint ← ∅
10.6      for {X, Y} ⊂ U do
10.7          if X ⊂ Y then
10.8              AddBlueArrow(G, X, Y)
10.9          else if X ⊃ Y then
10.10             AddBlueArrow(G, Y, X)
10.11         else if X ∦ Y then
10.12             AddGreenLine(G, X, Y)
10.13         else if X ∥ Y then
10.14             Nothing
10.15         else
10.16             AddGreenLine(G, X, Y)
10.17     for α ∈ G.nodes do
10.18         MaybeDisjointNode(G, α)
10.19     return G
```

Notes about Algorithm 10:

Line 10.2: *G.blue* will contain the set of blue arrows between certain nodes. Each element of *G.blue* is an ordered pair specifying the origin and destination of an arrow.

Line 10.3: *G.green* will contain the set of green lines between certain nodes. Each element of *G.green* is an unordered pair (with *set* semantics) of nodes.

Line 10.4: *G.nodes* is a set of labeled nodes. Initially *G.nodes* is the set of all nodes in the graph. The algorithm (or) continues until *G.nodes* is $\varnothing$. Each labeled node is a record with fields $\{label, subsets, supersets\}$. The label represents the Boolean expression for the subset in question. Originally this label is identically the element coming from U, but at the end of the algorithm, this label has become a Boolean combination of some elements of U.

Line 10.5: *G.disjoint* will contain the set of nodes, once they have become disjoint from all other nodes in the graph. Initially *G.disjoint* is $\varnothing$ but eventually, when *G.nodes* becomes $\varnothing$, *G.disjoint* will have become the complete set of disjoint nodes.

Line 10.6: Notice that $X \neq Y$ because $\{X, Y\}$ is a two element set.

Line 10.6: Notice that once $\{X, Y\}$ has been visited, $\{Y, X\}$ cannot be visited later, because it is the same set.

Line 10.16: This final `Else` clause covers the case in which it is not possible to determine whether $X \subset Y$ or whether $X \parallel Y$. In the worst case, they are non-disjoint, so draw a green line between X and Y.

Algorithm 11: MaybeDisjointNode

Input: G: a graph
Input: X: a node of G
Side Effects: Perhaps modifies $G.nodes$ and $G.disjoint$.

```
11.1 begin
11.2     if X.label = ∅ then
11.3         Delete X from G.nodes
11.4     else if ∅ = X.touches = X.supersets = X.subsets then
11.5         Delete X from G.nodes
11.6         Push X onto G.disjoint
```

Notes about Algorithm 11:

Line 11.2: Section 9.4.6 explains in more detail, but here we simply avoid collecting the empty set.

Line 11.6: This push should have set-semantics. *I.e.*, if there is already a set in $G.disjoint$ which is equivalent to this one, then do not push a second one.

Algorithm 14: AddBlueArrow

Input: G: a graph
Input: X: a node of G
Input: Y: a node of G
Side Effects: Modifies G adding a blue arrow from X to Y

```
14.1 begin
14.2     Push (X → Y) onto G.blue
14.3     Push X onto Y.subsets
14.4     Push Y onto X.supersets
```

Algorithm 15: DeleteBlueArrow

Input: G: a graph
Input: X: a node of G
Input: Y: a node of G
Side Effects: Modifies G removing the blue arrow from X to Y. Perhaps extends $G.disjoint$ and shortens $G.nodes$

```
15.1 begin
15.2     Remove (X → Y) from G.blue
15.3     Remove X from Y.subsets
15.4     Remove Y from X.supersets
15.5     MaybeDisjointNode(G, X)
15.6     MaybeDisjointNode(G, Y)
```

This construction assures that no two nodes have both a green line and a blue arrow between them.

There is an important design choice to be made: How to represent transitive subset relationships? There are two variations. We call these variations *implicit-inclusion vs. explicit-inclusion.*

The graph shown in Figure 9.4 uses explicit-inclusion. In the graph $A_5 \subset A_8 \subset A_1$. The question is whether it is necessary to include the blue arrow from ⑤ to ①. Explicit inclusion means that all three arrows are maintained in the graph: ⑤ to ⑧, ⑧ to ① and explicitly ⑤ to ①. Implicit-inclusion means that the arrow from ⑤ to ① is omitted.

The algorithm which we explain in this section can be made to accommodate either choice as long as the choice is enforced consistently. Omitting the arrow obviously lessens the number of arrows which have to be maintained, but makes part of the algorithm more tedious.

As far as the initialization of the graph is concerned, in the variation of implicit-inclusion, it is necessary to avoid creating arrows (or remove them after the fact) which are implicit.

9.4.2 Strict subset

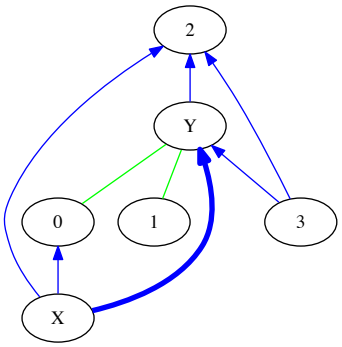
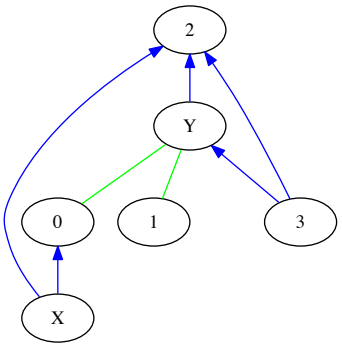
Before	After
	
Node X Y	**Relabeled Boolean expression** X $Y \cap \overline{X}$

Figure 9.5: Strict subset before and after mutation

Algorithm 16: BreakStrictSubset: Breaks a strict subset edge if possible

Input: G is a graph.
Input: X: a node in G
Input: Y: a node in G
Output: G
Side Effects: Possibly deletes a vertex and changes a label.

```
16.1  begin
16.2      if Y ∉ X.supersets then
16.3          Nothing
16.4      else if X.subsets ≠ ∅ then
16.5          Nothing
16.6      else if X.touches ≠ ∅ then
16.7          Nothing
16.8      else
16.9          Y.label ← Y ∩ ¬X
16.10         DeleteBlueArrow(G, X, Y)
16.11     return G
```

As shown in Algorithm 16, blue arrows indicate subset/superset relations; *i.e.* they point from a subset to a superset. A blue arrow from X to Y may be eliminated if the following conditions are met:

- X has no blue arrows pointing to it, and
- X has no green lines touching it.

These conditions mean that X represents a set which has no subsets elsewhere in the graph, and also that X represents a set which touches no other set in the graph.

There are as many as $\frac{8 \cdot 7}{2} = 28$ possible constraints imposed by pair relations. For each $\{X, Y\} \subset \{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8\}$:

Figure 9.5 illustrates this mutation. Node (Y) may have other connections, including blue arrows pointing to it or from it, and green lines connected to it. However node (X) has no green lines connected to it, and no blue arrows pointing to it; although it may have other blue arrows pointing away from it.

On eliminating the blue arrow, replace the label Y by $Y \cap \overline{X}$.

In Figure 9.4, nodes (5) and (6) meet this criteria. A node, such as (5) may have multiple arrows leaving it. Once the final such arrow is broken, the node becomes isolated.

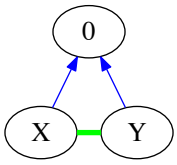


Figure 9.6: Subtle case

The restriction, that the node X have no green line touching it, is subtle. Consider the graph in Figure 9.6. If either the blue arrow from Ⓧ to ⓪ or the blue arrow from Ⓨ to ⓪ is broken by the rule *Strict subset*, then the other of the two arrows becomes incorrect. Therefore, we have the restriction for the *Strict subset* rule that Ⓧ and Ⓨ have no green lines connecting to them. The *Relaxed subset* condition is intended to cover this case.

In the variation of implicit-inclusion it is necessary to add all the superclasses of Y to become also superclasses of X. This means for each blue arrow from Ⓨ to ⓝ in the graph, we must add blue arrows leading from Ⓧ to ⓝ. In the example 9.5, if we were using implicit-inclusion, the arrow from Ⓧ to ② would be missing, as it would be implied by the other arrows. Therefore, the blue arrow from Ⓧ to ② would need to be added in the *After* graph of Figure 9.5.

9.4.3 Relaxed subset

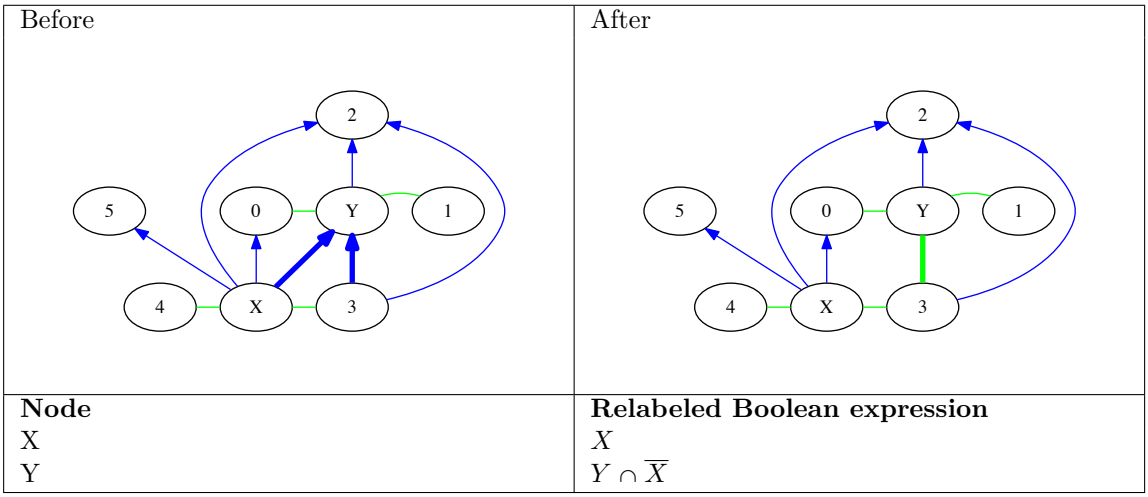


Figure 9.7: Relaxed subset before and after mutation. The blue arrow from X to Y has been eliminated, and the blue arrow from 3 to Y has been converted to a green line because X and 3 are connected with a green line.

Algorithm 17 is similar to the *Strict subset* algorithm except that the subset node X is allowed to touch other nodes. But special attention is given if X touches a sibling node; *i.e.* if X has a green line connecting it to an intermediate node which also has a blue arrow pointing to Y. This case is illustrated in the *Before* graph in Figure 9.7. This graph can be mutated to the *After* graph shown in Figure 9.7.

The node label transformations for this case are exactly the same as those for the *Strict subset* condition. The only different consequence is that sometimes a blue arrow must be transformed into a green line. Each blue arrow connecting a node to Ⓨ (such as one connecting ③ → Ⓨ in before graph *Before* graph of Figure 9.7) which is also connected to X by a green line must be transformed from a blue arrow to a green line as shown in the *After* graph.

In the variation of implicit-inclusion it is necessary to add blue arrows representing all the transitive superclass relations. This means in the graph, for each blue arrow from Ⓨ to ⓝ, we must add blue arrows leading from Ⓧ to ⓝ and also from ③ to ⓝ. In the example 9.7, if we were using implicit-inclusion, the arrows from Ⓧ to ② and from ③ to ② would be missing as they would be implied by the other arrows. Therefore, the blue arrows from Ⓧ to ② and from ③ to ② would need to be added in the *After* graph.

Algorithm 17: BreakRelaxedSubset Breaks a subset relation and some other subset relations to the common superset

Input: G: a graph
Input: X: a node in G
Input: Y: a node in G
Output: G
Side Effects: Perhaps changes a label, and some blue vertices removed or converted to green.

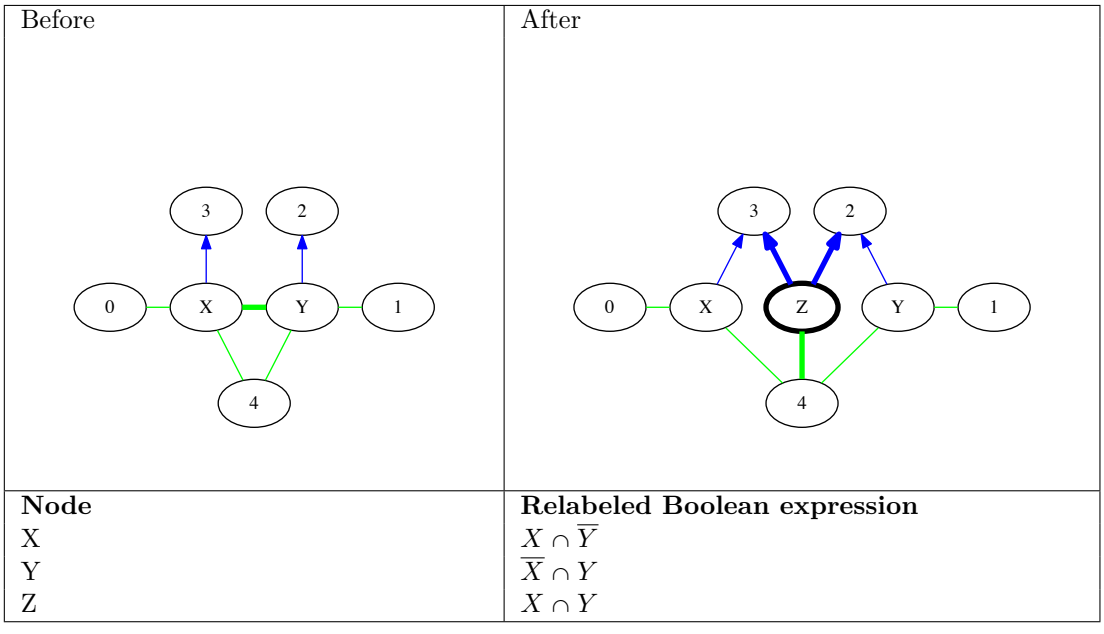
```
17.1  begin
17.2      if Y ∉ X.supersets then
17.3          Nothing
17.4      else if X.subsets ≠ ∅ then
17.5          Nothing
17.6      else
17.7          Y.label ← Y ∩ ¬X
17.8          for α ∈ (X.touches ∩ Y.subsets) do
17.9              AddGreenLine(G, α, Y)
17.10             DeleteBlueArrow(G, α, Y)
17.11         DeleteBlueArrow(G, X, Y)
17.12     return G
```

Notes about Algorithm 17:

Line 17.8: α iterates over the intersection of $X.touches$ and $Y.subsets$.

Line 17.10: Be careful to *add* and *delete* in that order. Reversing the order may cause the function $DeleteBlueArrow$ to mark the node as disjoint via a call to $MaybeDisjointNode$.

9.4.4 Touching connections



Node	Relabeled Boolean expression
X	$X \cap \overline{Y}$
Y	$\overline{X} \cap Y$
Z	$X \cap Y$

Figure 9.8: Touching connections before and after mutation

Green lines indicate partially overlapping sets. A green line connecting X and Y may be broken if the following condition is met:

- Neither X nor Y has a blue arrow pointing to it; *i.e.* neither represents a superset of something else in the graph.

Eliminating the green line *separates* X and Y. To do this X and Y must be relabeled and a new node must be added to the graph. Algorithm 18 explains this procedure.

Algorithm 18: BreakTouching

Input: G: a graph
Input: X: a node in G
Input: Y: a node in G
Output: G
Side Effects: Perhaps removes some vertices from G, and adds new nodes and vertices.

```
18.1  begin
18.2      if Y ∉ X.touches then
18.3          Nothing
18.4      else if X.subsets ≠ ∅ then
18.5          Nothing
18.6      else if Y.subsets ≠ ∅ then
18.7          Nothing
18.8      else
18.9          Z ← G.AddNode()
18.10         Z.label ← X ∩ Y
18.11         (X.label, Y.label) ← (X ∩ Ȳ, Y ∩ X̄)
18.12         for α ∈ (X.supersets ∪ Y.supersets) do
18.13             AddBlueArrow(G, Z, α)
18.14         for α ∈ (X.touches ∩ Y.touches) do
18.15             AddGreenLine(G, Z, α)
18.16         DeleteGreenLine(G, X, Y)
18.17         MaybeDisjointNode(G, Z)
18.18     return G
```

Notes about Algorithm 18:

Line 18.11: This is a parallel assignment. *I.e.*, calculate $X \cap \overline{Y}$ and $Y \cap \overline{X}$ before assigning them respectively to $X.label$ and $Y.label$.

Line 18.10: Introduce new node labeled $X \cap Y$.

Line 18.12: Blue union; draw blue arrows from this node, $X \cap Y$, to all the nodes which either X or Y points to. *I.e.*, the supersets of $X \cap Y$ are the union of the supersets of X and of Y.

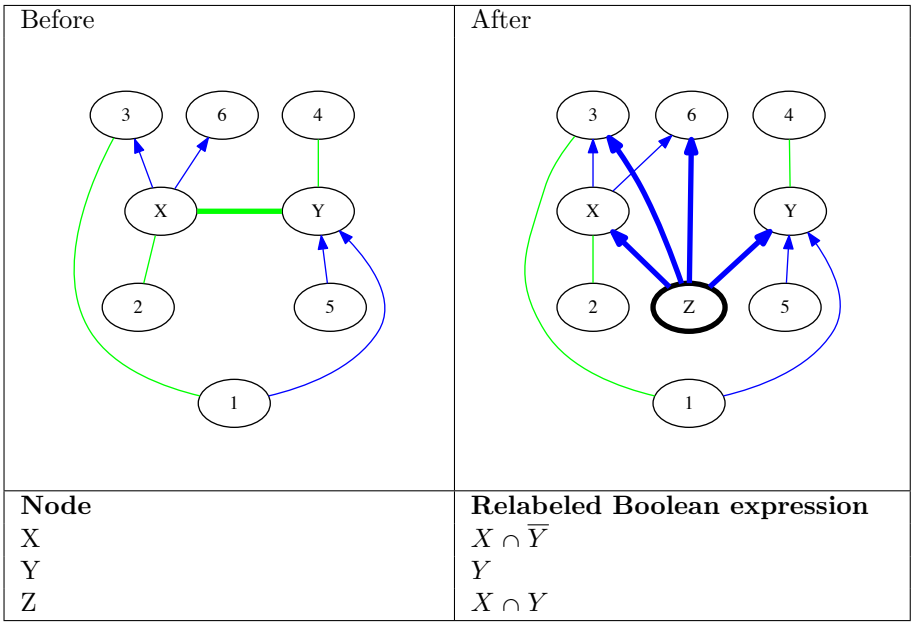
Line 18.14: Green Intersection; draw green lines from $X \cap Y$ to all nodes which both X and Y connect to. *I.e.* the connections to $X \cap Y$ are the intersection of the connections of X and of Y.

Line 18.15: Exception, if there is (would be) a green and blue vertex between two particular nodes, omit the green one.

Line 18.16: Be careful to *add* and *delete* in that order. Calling *DeleteGreenLine* before *AddGreenLine* cause the function *DeleteGreenLine* to mark the node as disjoint via a call to *MaybeDisjointNode*.

Figure 9.8 illustrates the step of breaking such a connection between nodes Ⓧ and Ⓨ by introducing the node Ⓩ.

9.4.5 Loops



Node	**Relabeled Boolean expression**
X	$X \cap \overline{Y}$
Y	Y
Z	$X \cap Y$

Figure 9.9: Graph meeting the *loop* condition

This step can be omitted if one of the previous conditions is met: either *strict subset* or *touching connection*. The decision of whether to omit this step is not a correctness question, but rather a performance question which is addressed in Section 10.5.

As is detailed in Algorithm 19, a green line connecting Ⓧ and Ⓨ may be removed if the following conditions are met:

- Ⓧ has no blue arrow pointing to it.
- Ⓨ has at least one blue arrow pointing to it.

The rare necessity of this operation arises in graphs such as the green line connecting Ⓧ and Ⓨ in Figure 9.9. To eliminate this green line, proceed by splitting node Ⓧ into two nodes: the part that is included in set Y and the part that is disjoint from set Y. The result of the mutation is shown in the *After* graph.

- Remove the green line between Ⓧ and Ⓨ.
- Create a new node Ⓩ copying all the green and blue connections from Ⓧ.
- Create a blue arrow from Ⓩ to Ⓨ, because $Z = X \cap Y \subset Y$.
- Create a blue arrow from Ⓩ to Ⓧ, because $Z = X \cap Y \subset X$.
- $Z \leftarrow X \cap Y$; *i.e.* label the new node.
- $X \leftarrow \overline{X} \cap Y$; *i.e.* relabel Ⓧ.
- For each node ⓝ, which is a parent of either Ⓧ or Ⓨ (union), draw a blue arrow from Ⓩ to ⓝ.

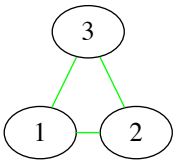
This graph operation effectively replaces the two nodes X and Y with the three nodes $X \cap Y$, $X \cap \overline{Y}$, and Y. This is a reasonable operation because $X \cup Y = X \cap \overline{Y} \cup \overline{X} \cap Y \cup Y$.

Algorithm 19: BreakLoop

Input: G: a graph
Input: X: a node in G
Input: Y: a node in G
Output: G
Side Effects: May remove some of its green vertices, and adds new nodes and blue vertices.

```
19.1  begin
19.2      if Y ∉ X.touches then
19.3          Nothing
19.4      else if X.subsets ≠ ∅ then
19.5          Nothing
19.6      else if Y.subsets = ∅ then
19.7          Nothing
19.8      else
19.9          Z ← G.AddNode()
19.10         Z.label ← X ∩ Y
19.11         X.label ← X ∩ ¬Y
19.12         for α ∈ X.touches do
19.13             AddGreenLine(G, Z, α)
19.14         for α ∈ (X.supersets ∪ Y.supersets) do
19.15             AddBlueArrow(G, Z, α)
19.16         AddBlueArrow(G, Z, Y)
19.17         AddBlueArrow(G, Z, X)
19.18         DeleteBlueArrow(G, X, Y)
19.19     return G
```

9.4.6 Discovered empty set



Name	Definition
A_1	$\{1, 2, 3, 4\}$
A_2	$\{1, 2, 3, 5\}$
A_3	$\{1, 2, 3, 6\}$

Figure 9.10: Setup for Discovered Empty Set

During the execution of the algorithm, it may happen that derived sets are discovered to be empty. This occurrence is a consequence of the semantics of the green lines and blue arrows. Recall that a green line connecting two nodes indicates that the corresponding sets are not *known* to be disjoint. A consequence of this semantic is that two nodes representing sets which are not yet known to be disjoint are connected by a green line. When the intersection of the two sets is eventually calculated, the intersection will be found to be empty.

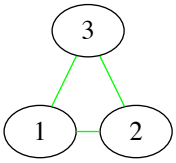
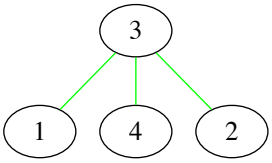
Before	After
	
Node	**Relabeled Set**
$A_1 = \{4\}$	$A_3 \cap \overline{A_2}$
$A_2 = \{5\}$	$\overline{A_3} \cap A_2$
$A_3 = \{1, 2, 3, 6\}$	A_3
$A_4 = \{1, 2, 3\}$	$A_1 \cap A_2$

Figure 9.11: Connection between disjoint sets

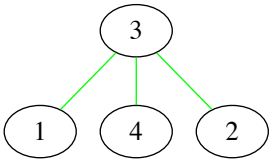
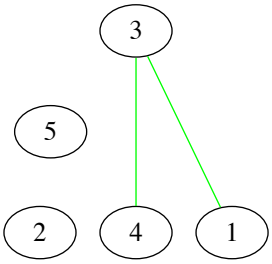
Before	After
	
Node	**Relabeled Set**
$A_1 = \{4\}$	A_1
$A_2 = \{5\}$	$A_2 \cap \overline{A_3}$
$A_3 = \{1, 2, 3, 6\}$	$\overline{A_2} \cap A_3$
$A_4 = \{1, 2, 3\}$	A_4
$A_5 = \varnothing$	$A_2 \cap A_3$

Figure 9.12: Discovered Empty Set

Example 9.4 (Discovered empty set)**.** Consider the definitions and graph shown in Figure 9.10. If we break the connection between node ① and ②, the configuration in Figure 9.11 results. The resulting graph shows three green lines, connecting ① and ③, connecting ② and ③, and connecting ④ and ③.

The next step does not cause a problem in the semantics of the representation, but rather it is a performance issue. Figure 9.12 shows the result of breaking the green line connecting ② and ③. Node ② becomes isolated which correctly represents the set $\{5\}$, and ⑤ is derived which represents $\varnothing$.

The semantics are preserved because $\bigcup_{n=1}^{5} A_n = \{1, 2, 3, 4, 5\}$, and the isolated nodes ② and ⑤ represent disjoint sets: $\{5\} \parallel \varnothing$. However, there is a potential performance issue, because it is possible that some of the isolated sets are empty. At some point a vacuity check must be made on these derived sets.

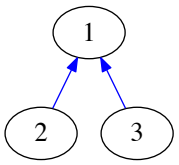
	Name	Definition
	A_1	$\{1, 2, 3, 4\}$
	A_2	$\{1, 2, 3\}$
	A_3	$\{4\}$

Figure 9.13: Setup for Discovered Equal Sets

Another case where a set may be discovered to be empty is when the relative complement is performed between a superset and subset. If the two sets are actually equal (recall that $A = B \implies A \subset B$) then the relative complement is empty, ($A = B \implies A \cap \overline{B} = \varnothing$).

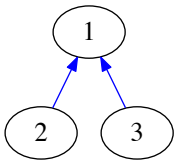
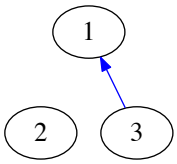
Before	After
	
Node $A_1 = \{4\}$ $A_2 = \{1, 2, 3\}$ $A_3 = \{4\}$	**Relabeled Set** $A_1 \cap \overline{A_2}$ A_2 A_3

Figure 9.14: Connection between equal sets

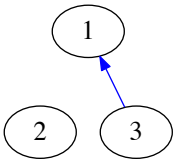
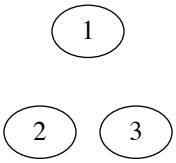
Before	After
	
Node $A_1 = \varnothing$ $A_2 = \{1, 2, 3\}$ $A_3 = \{4\}$	**Relabeled Set** $A_1 \cap \overline{A_3}$ A_2 A_3

Figure 9.15: Connection between equal sets

Example 9.5 (Situation leading to isolated nodes representing $\varnothing$.)**.** Consider the definitions and graph shown in Figure 9.13. In Figure 9.14 we break the blue arrow from ② to ①, then a blue arrow remains from ③ to ①, which represents a set which is a subset of itself. As in Example 9.4, this is not an error in the semantics of the representation, but rather a performance problem. In Figure 9.15 we break the connection from ③ to ① resulting in three isolated nodes representing the sets $\varnothing$, $\{1, 2, 3\}$, and $\{4\}$ which are three disjoint sets whose union is $\{1, 2, 3, 4\}$, as expected.

Since intersection and relative complement operations may result in the empty set, as illustrated in Examples 9.4 and 9.5, the implementation of the set disjoint decomposition algorithm must take this possibility into

account. There are several possible approaches.

Late check: The simplest approach is to check for vacuity once a node has been disconnected from the graph. Once a node has no touching nodes, no subset nodes, and no superset nodes, it should be *removed* from the graph data structure, checked for vacuity, and then discarded, if empty.

Correct label: Test for $\varnothing$ each time a label of a node changes, *i.e.*, it thereafter represents a smaller set than before.

Correct connection: Each time the label of a node changes, *i.e.* it thereafter represents a smaller set than before, all of the links to neighbors (green lines and blue arrows) become suspect. *I.e.*, when a label changes, re-validate all the touching, subset, and superset relationships and update the blue arrows and green lines accordingly.

The choice of data structure used for the algorithm may influence how expensive the vacuity check is, and thus may influence which approach should be taken.

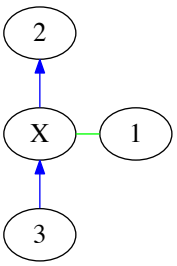


Figure 9.16: Discovered empty set.

When a node, such as X in Figure 9.16, is discovered to represent the empty set, its connections must be visited.

Touching node: A green line connecting the node to another node can be deleted, because the empty set is disjoint from every set including from the empty set. In Figure 9.16 if X is discovered to represent $\varnothing$, then the green line between Ⓧ and ① can simply be deleted.

Superset: A blue arrow from the node to another node may be deleted. In Figure 9.16 if X is discovered to represent $\varnothing$, then the blue arrow from Ⓧ to ② can be deleted. This is because if we attempted to relabel A_2 as $A_2 \cap \overline{X}$ we'd result again with A_2 because $A_2 \backslash \varnothing = A_2$.

Subset: A blue arrow from another node to the node in question can be removed, and the other node can be inferred to represent the empty set. Thus the empty set reduction can be applied recursively to that node. In Figure 9.16 if X is discovered to represent $\varnothing$, then we know that $A_3 \subset \varnothing$ and thus $A_3 = \varnothing$. This means we can delete the blue arrow from ③ to Ⓧ, and then recursively apply the reduction to ③.

9.4.7 Recursion and order of iteration

It is natural to wonder whether the order in which the nodes are visited may have an effect on the execution time of the algorithm. The manner in which the traversal order affects the calculation performance has been investigated and is explained in Section 10.5.

Our implementation represents nodes of the graph as objects, where each node object has three slots containing lists of subset, superset, and touching nodes. Thus graph reduction, in our implementation, involves iterations over the nodes until they loose all their connections.

An alternate implementation might as well represent connections as objects, thus allowing the reduction algorithm to directly iterate over the connections until they are eliminated. We have not investigated this approach.

Given that our algorithm is required to visit each node, there are several obvious strategies to choose the order of iteration. This boils down to sorting the list of nodes into some order before iterating over it. We have investigated five such strategies.

SHUFFLE:	Arrange the list into random order.
INCREASING-CONNECTIONS:	Sort the nodes into order of increasing number of connections; *i.e.*, number of touching nodes plus the number of subset nodes plus the number of superset nodes.
DECREASING-CONNECTIONS:	Sort the nodes into order of decreasing number of connections.
BOTTOM-TO-TOP:	Sort into increasing order according to number of superset nodes. This sort assumes supersets are explicitly represented in the connections, because a subset node will also contain connections to the supersets of its direct supersets.
TOP-TO-BOTTOM:	Sort into decreasing order according to number of superset nodes.

All of the graph operations described in Section 9.4 depend on a node not having subset nodes. An additional detail of the strategy, which can be (and has been) employed in addition to the node visitation order, is whether to recursively apply the reduction attempts to subset nodes before superset nodes. *I.e.*, while iterating over the nodes, would we recursively visit subset nodes? The hope is that a node is more likely to be isolate-able if we attempt to break subclass relations directly when encountered, rather than treating the nodes when they appear in the visitation order.

9.5 Type decomposition using BDDs

See Chapter 5 for a discussion of Binary Decision Diagrams.

Using BDDs in these algorithms allows certain checks to be made more easily than with the s-expression approach. For example, two types are equal if they are the same object (pointer comparison, `eq`). A type is empty if it is identically the empty type (pointer comparison). Finally, given two types (represented by BDDs), the subtype check can be made using the function in Implementation 9.6.

Implementation 9.6 (`bdd-subtypep`)**.**

```
(defun bdd-subtypep (bdd-sub bdd-super)
  (eq *bdd-false*
      (bdd-and-not bdd-sub bdd-super)))
```

This implementation of `bdd-subtype` should not be interpreted to mean that we have obviated the need for the Common Lisp `subtypep` function. In fact, `subtypep` is still useful in constructing the BDD itself. However, once the BDDs have been constructed and cached, subtype checks may at that point avoid calls to `subtypep`, which, in some cases, might otherwise be more computationally intensive.

To decompose a set of types using the BDD approach we start with the list of type specifiers, eliminate the ones which specify the empty type, and proceed as follows.

Seed the set, S, with one of the types. Iterate t_1 through the remaining types, represented as BDDs. For each t_1, iterate t_2 through the elements of S. Calculate a slice–set of at most three elements by calculating $\{t_1 \cap t_2,\ t_1 \cap \overline{t_2},\ \overline{t_1} \cap t_2\} \cap \overline{\{\varnothing\}}$, discarding any that are the empty type and accumulating the non-empty types. After t_2 has traversed completely through S, replace S by the union of the slice–sets, and proceed with the next value of t_1. When calculating this union of slice–sets, it is important to guard against duplicate elements as some slice–sets may contain common elements. After t_1 finishes iterating through the given set of types, S remains the type decomposition.

The Common Lisp function in Implementation 9.7 is coded elegantly using the `reduce` function.

There are a couple of subtle points about the algorithm. At `NOTE-A`, we ask whether the intersection of A and B is the empty type ($A \parallel B$), because then we don't need to calculate $A \backslash B$ and $B \backslash A$ as we know that $A \backslash B = A$ and $B \backslash A = B$. Thus we simply add B to the set being accumulated.

At `NOTE-B`, we have discovered that intersection of A and B is non empty ($A \nparallel B$), so we augment the set with at most 3 types. Looking at $\{A \cap B, A \backslash B, B \backslash A\} \backslash \{\varnothing\}$; we remove duplicates as some of them might be equal. In this case we provide `nil` as the value for `all-disjoint?` in the next iteration of `reduce`, because we've found something A is not disjoint with.

Implementation 9.7 (mdtd-bdd).

```
(defun mdtd-bdd (type-specifiers)
  (let ((bdds (remove-if #'bdd-empty-type (mapcar #'ltbdd type-specifiers))))
    (labels
        ((try (bdds disjoint-bdds &aux (bdd-a (car bdds)))
           (cond
             ((null bdds)
              disjoint-bdds)
             (t
              (flet
                  ((reduction (acc bdd-b &aux (bdd-ab (bdd-and bdd-a bdd-b)))
                     (destructuring-bind (all-disjoint? bdd-set) acc
                       (cond
                         ((bdd-empty-type bdd-ab) ;; NOTE-A
                          (list all-disjoint? (adjoin bdd-b bdd-set)))
                         (t ;; NOTE-B
                          (list nil
                                (union (remove-duplicates
                                        (remove-if #'bdd-empty-type
                                                   (list bdd-ab
                                                         (bdd-and-not bdd-a bdd-ab)
                                                         (bdd-and-not bdd-b bdd-ab))))
                                       bdd-set)))))))
                (destructuring-bind (all-disjoint? bdd-set)
                    (reduce #'reduction (cdr bdds) :initial-value '(t nil))
                  (try bdd-set
                       (if all-disjoint?
                           (pushnew bdd-a disjoint-bdds)
                           disjoint-bdds)))))))))
      (try bdds nil))))
```

Using BDDs in this algorithm allows certain checks to be made easily. For example, two types are equal if they are the same object (pointer comparison). A type is empty if it is identically the empty type (pointer comparison). Two BDDs which share equal common subtrees, actually share the objects (shared pointers).

9.5.1 Improving the baseline algorithm using BDDs

We may revisit the algorithms described in Sections 9.1, 9.4, and 9.3, but this time use the BDD as the data structure to represent the Common Lisp type specifications rather than using the s-expression.

The algorithm (see Appendix C) initializes one variable, U, to the given set of types, and another variable, D, to the empty set. A loop is repeated until U is the empty set, at which time D is the set of disjoint types.

Each iteration through the loop involves a single call to Common Lisp `reduce`. Each call to reduce chooses one fixed element, called A, and `reduce` iterates B over the rest of the set. At each iteration, the intersection $A \cap B$ is calculated. If $A \cap B = \varnothing$, then $V = V \cup \{B\}$, else $V = V \cup \{A \cap B, A \backslash B, B \backslash A\} \backslash \{\varnothing\}$. If after the iteration (*i.e.*, after `reduce` returns), it is discovered that $A \cap B$ was empty for each value of B, then accumulate $D = D \cup \{A\}$. And in either case set $U = V$ and repeat the loop until U becomes empty.

9.5.2 Improving the graph-based algorithm using BDDs

We re-implemented the graph algorithm described in Section 9.4. The implementation is roughly 200 lines of generic CLOS Common Lisp code plus 30 lines of BDD specific code–roughly 5 times the size of the brute force algorithm described in Section 9.1. No doubt, this algorithm could be written more concisely, but the CLOS implementation easily allows both the s-expression based and the BDD based code to use the same base code. The differences exist only in the code pertaining to the classes `sexp-node` and `sexp-graph` in the case of the s-expression based solution and `node-of-bdd` and `bdd-graph` in the case of the BDD based solution.

The skeleton of the code, `mdtd-bdd-graph`, is straightforward and is shown in Appendix D including subsection D.12.

9.6 The Baker `subtypep` implementation

Baker [Bak92] noted that some implementations of Common Lisp have inefficient implementations of the `subtypep` [Ans94] function which are both slow and also fail to determine some subtype relations which are computable. The `subtypep` function is permitted to return `nil,nil` in certain cases, indicating *don't-know*. The most obvious such case is when the subtype relation is genuinely uncomputable in some cases involving `satisfies`; *e.g.*, `(subtype '(satisfies F) '(satisfies G))`. However, some subtype questions involving `satisfies` are computatable; *e.g.* `(subtypep '(and number (satisfies F)) '(satisfies F))`.

Another reason Common Lisp implementations chose to return *don't-know* is when the calculation might take an excessively long time to calculate. Such abuse is permitted by the Common Lisp specification in cases involving `and`, `eql`, the list form of `function`, `member`, `not`, `or`, `satisfies`, or `values`.

Baker outlined an efficient procedure which he claims computes the subtype relation in every case where such computation is possible. Valais [Val18] implemented a subset of the Baker procedure in SBCL. Even though the implementation is incomplete, and not yet thoroughly tested, we made a very simple implementation function for testing. The functions `mdtd-bdd-graph-baker` and `mdtd-graph-baker` implement the BDD and non BDD version of the algorithm explained in Section 9.5.2. The only difference between the Baker and non-Baker versions are the binding of the global variable `*subtypep*` to either `baker:subtypep` or `cl:subtypep`.

Chapter 10

Performance of MDTD Algorithms

In Chapter 9, Sections 9.1, 9.4, and 9.3 explained three different algorithms for calculating type decomposition. Now we look at some performance characteristics of variants of the algorithms. Section 10.1 describes what the tests include. Section 10.2 describes content and construction of data sets, which we call pools, used in the tests. Section 10.4 describes experiments to determine which hash table strategy to use, based on the run-time performance of each. Section 10.5 describes how the tuning parameters were selected for the optimized BDD based graph algorithm. Section 10.6 describes the relative performance of the baseline algorithm from Section 9.1 using (1) its s-expression based implementation and also (2) its BDD based implementation; the graph based algorithm from Section 9.5.2 including (3) its s-expression based implementation, (4) its BDD based implementation and (5) the SAT like algorithm from Section 9.3.

10.1 Overview of the tests

The performance tests start with a list of type specifiers, randomly selected from a pool, each time with a randomly select number of type specifiers, and then call one or more of the functions to calculate the MDTD and record the time of each calculation or, in some cases, record the profiler data. We have plotted results of the runs which completed, as opposed to those which timed out. This omission of data points of timed-out experiments does not, in any way, affect the presentation of which algorithms were the fastest on each test.

We have shared access to a cluster of Intel Xeon^{TM} E5-2620 2.00GHz 256GB DDR3 machines. The tests were performed on this cluster, using SBCL 1.3.0 ANSI Common Lisp.

We attempted to plot the results many different ways: time as a function of input size, number of disjoint sets in the input, number of new types generated in the output. The plot which we found heuristically to show the strongest visual correlation was 'calculation time *vs.* the product size'.

Definition 10.1. When running a single MDTD algorithm to calculate a disjoint set of types of cardinality, u, given a possibly overlapping set of type specifiers of cardinality v, we define the product size as the integer product of the number of given input types multiplied by the number of calculated output types.

$$\text{product size } = u \cdot v$$

Example 10.2 (Calculation of product size)**.**
If an MDTD algorithm takes a list of $v = 5$ type specifiers and computes $u = 3$ disjoint types in 0.1 seconds, the plot contains a point at $(u \cdot v, 0.1) = (15, 0.1)$.

Although we don't claim to completely understand why this particular plotting strategy shows better correlation than the others we tried, it does seem that all the algorithms begin a $\mathcal{O}(n^2)$ loop by iterating over the given set of types which is incrementally converted to the output types. So the algorithms, in some sense, finish by iterating over the output types. More research is needed to better understand the correlation.

The plots are generally subject to a high degree of noise, so in this report we have chosen to display smoothened renditions of the plots. The plots in Figure 10.1 show an example of data actually measured and the same data after the smoothing procedure was run. In these particular plots we display the calculation time

of five variants of the algorithms *vs.* product size (Definition 10.1). The given set of type specifiers are taken from the pool, **subtypes of number** (Section 10.2.1).

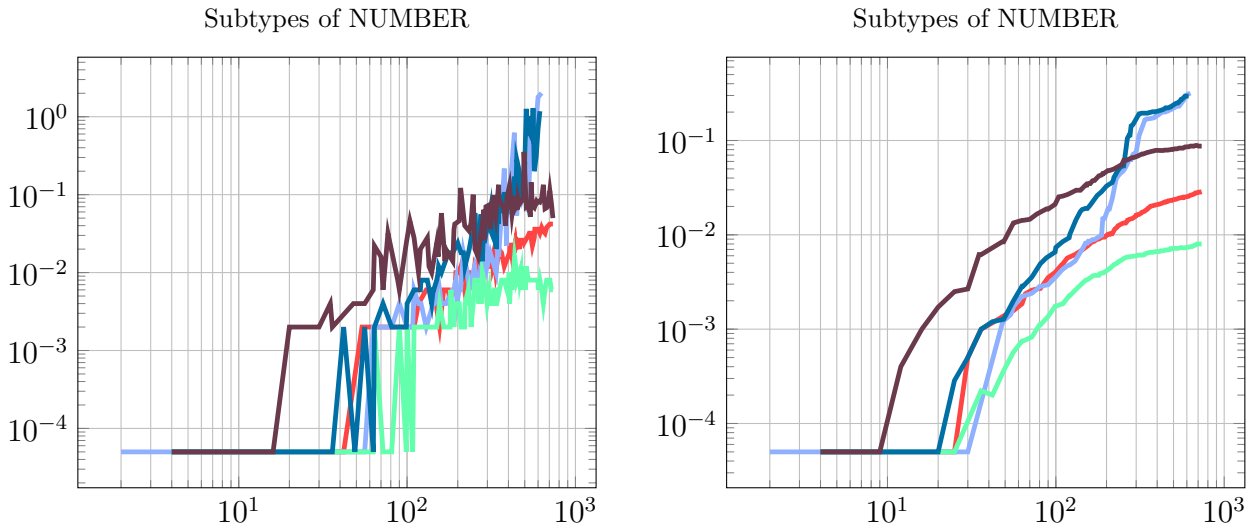


Figure 10.1: Run-time performance of MDTD algorithms applied to the pool consisting of subtypes of number. The plots show the measured data (left) and smoothened data (right). Each plot is displayed with y='Computation Time (seconds)' *vs.* x='Product Size'.

The algorithm for smoothening is as follows. The y-coordinate of each point, (x_0, y_0) is replaced by the arithmetic mean of the y-coordinates of all the points whose x-coordinate, x is within the range $\frac{x_0}{2} \leqslant x \leqslant 2 \cdot x_0$, as shown in Implementation 10.3.

Implementation 10.3 (`smoothen`).

```
(defun smoothen (xys &key (radius 2))
  (flet ((mean (xs)
           (/ (reduce #'+ xs :initial-value 0.0)
              (float (length xs))))
         (x-coord (point)
           (car point))
         (y-coord (point)
           (cadr point)))

    (loop :for xy :in xys
          :for x0 = (car xy)
          :for close = (loop :for pt :in xys
                             :when (<= (/ x0 radius)
                                       (x-coord pt)
                                       (* x0 radius))
                               :collect (y-coord pt))
          :collect (list x0 (mean close)))))
```

Other types of means (such as a geometric mean) and other radii are, of course, possible. We did not experiment with other approaches because we only use the smooth plots for display purposes. All meaningful calculations pertaining to the plots were performed with the measured data.

10.2 Pools of type specifiers used in performance testing

In this section, Section 10.2, the reader may notice that although we consider many different kinds of Common Lisp type specifiers, we conspicuously ignore function types such as `(function (string) integer)`. This omission is not accidental. In Section 2.7 we explain several issues related to function types in Common Lisp.

We made the explicit decision, in light of these problems, to omit consideration of function types from our testing of the MDTD algorithms.

It has been observed that different sets of initial types evoke different performance behavior from the various algorithms. Further research is needed to explain and characterize this behavior. It would be ideal if, from a given set of types, it were possible to predict, with some amount of certainty, the time required to calculate the type decomposition. Currently, this is not possible. What we have done instead is put together several sets of types which can be used as *pools* or starting sets for the decomposition.

The pools used in the experiments and whose results are illustrated in Figure 10.8 are explained in the following sections.

10.2.1 Pool: Subtypes of `number`

This pool contains the type specifiers for all the subtypes of `cl:number` whose symbol name comes from the `"CL"` package:

```
(array-total-size float-radix ratio rational bit nil array-rank real
 double-float bignum signed-byte float unsigned-byte
 single-float char-code number float-digits fixnum complex)
```

In SBCL these are all type specifiers, even though they are not explicitly denoted as such in the Common Lisp specification.

The list of type specifiers was determined using reflection. The function, `valid-subtypes` shown in Implementation 10.4 was used. A call to `(valid-subtypes 'number)` returns the list of type specifiers shown above. The `valid-subtypes` relies on a helper function, `valid-type-p` also shown in Implementation 10.4.

Implementation 10.4 (Code for `valid-subtypes`).

```
(defun valid-subtypes (super &key (test (lambda (t1 t2)
                                          (and (subtypep t1 t2)
                                               (subtypep t2 t1)))))
  (let (all-types)
    (do-external-symbols (sym :cl)
      (when (and (valid-type-p sym)
                 (subtypep sym super))
        (push sym all-types)))
    (remove-duplicates all-types :test test)))

(defun valid-type-p (type-designator)
  "Predicate to determine whether the given object is a valid type specifier."
  #+sbcl (handler-case (and (SB-EXT:VALID-TYPE-SPECIFIER-P type-designator)
                            (not (eq type-designator 'cl:*)))
           (SB-KERNEL::PARSE-UNKNOWN-TYPE (c) (declare (ignore c)) nil))
  #+(or clisp allegro)
    (ignore-errors (subtypep type-designator t))
  #-(or sbcl clisp allegro)
    (error "VALID-TYEP-P not implemented for ~A" (lisp-implementation-type))
  )
```

10.2.2 Pool: Subtypes of `condition`

This pool contains the type specifiers for all the subtypes of `cl:condition` whose symbol name comes from the `"CL"` package:

```
(simple-error storage-condition file-error control-error serious-condition
 condition division-by-zero nil parse-error simple-type-error error
 package-error program-error stream-error unbound-variable undefined-function
 floating-point-inexact cell-error floating-point-overflow
 floating-point-invalid-operation simple-warning print-not-readable type-error
 floating-point-underflow style-warning end-of-file unbound-slot reader-error
 simple-condition arithmetic-error warning)
```

This list was generated with a call to `(valid-subtypes 'condition)`. The code for `valid-subtypes` can be found in Implementation 10.4.

10.2.3 Pool: Subtypes of `number` or `condition`

This pool contains the type specifiers for all the subtypes of `cl:number` and all the subtypes of `cl:condition` whose symbol name comes from the `"CL"` package. It is the union of the two sets from Section 10.2.1 and 10.2.2.

10.2.4 Pool: Real number ranges

This pool contains the type specifiers of ranges of real numbers. Each type specifier is a range of `float`, `real`, or `integer`. In each case the lower or upper bound may be included or excluded. Example values are as follows:

```
((real 1/4 5/4) (real (1/4) 5/4) (real 1/4 (5/4)) (real (1/4) (5/4))
 (real 62/33 41/20) (real (62/33) 41/20) (real 62/33 (41/20))
 (float 63.6 98.51) (float (63.6) 98.51) (float 63.6 (98.51))
 (float (63.6) (98.51)) (integer 20 98) (integer (20) 98)
 (integer 20 (98)) (integer (20) (98))
 ...)
```

10.2.5 Pool: Integer ranges

This pool contains the type specifiers which are all ranges of integers, some `fixnum` and sum `bignum`. Example values are as follows:

```
((integer (0) 72) (integer 75 (88)) (integer (45174) (334427))
 (integer (445) 7536172) (integer (4013) (53830)) (integer (0) 69)
 (integer (301) (1474407)) (integer (542801460681) 9207612574201)
 (integer (925) 93734076688)
...)
```

10.2.6 Pool: Subtypes of `cl:t`

This pool contains the `"CL"` package symbols specifying a subtype of `cl:t`.

```
(simple-error storage-condition file-error array-total-size float-radix ratio
 character restart package rational control-error t vector method
 serious-condition atom generic-function condition bit readtable
 division-by-zero nil parse-error null base-string base-char simple-type-error
 synonym-stream error stream package-error array-rank pathname-host
 standard-object simple-base-string keyword boolean program-error file-stream
 stream-error unbound-variable sequence undefined-function real cons
 floating-point-inexact double-float concatenated-stream bit-vector
 standard-method cell-error floating-point-overflow hash-table
 method-combination floating-point-invalid-operation simple-warning bignum
 signed-byte compiled-function float array unsigned-byte single-float symbol
 pathname-device char-code print-not-readable type-error function simple-array
 floating-point-underflow simple-string number simple-bit-vector style-warning
 standard-char echo-stream standard-class logical-pathname float-digits
 structure-object pathname-version two-way-stream fixnum built-in-class
 end-of-file unbound-slot extended-char reader-error string-stream pathname
 random-state standard-generic-function simple-condition class list
 structure-class arithmetic-error pathname-type broadcast-stream warning
 complex simple-vector string)
```

This list was generated with a call to `(valid-subtypes cl:t)`. The code for `valid-subtypes` can be found in Implementation 10.4. As before, some of the symbols in this list, `char-code` and `float-digits` for example, are indeed type specifiers in SBCL, even if they are not specified to be so in the to Common Lisp specification.

10.2.7 Pool: Subtypes in `SB-PCL`

This pool contains the type specifiers for all the types whose symbol is in the `"SB-PCL"` package:

```
(sb-mop:slot-definition sb-mop:standard-slot-definition sb-mop:specializer
 sb-mop:funcallable-standard-class sb-mop:standard-reader-method
 sb-mop:standard-writer-method sb-pcl:system-class sb-mop:eql-specializer
 sb-mop:standard-accessor-method sb-mop:forward-referenced-class
 sb-mop:standard-direct-slot-definition sb-mop:direct-slot-definition
 sb-mop:effective-slot-definition sb-mop:funcallable-standard-object
 sb-mop:standard-effective-slot-definition)
```

10.2.8 Pool: Specified Common Lisp types

This pool contains the symbols naming a Common Lisp specified type. These are the 97 types listed in **Figure 4-2. Standardized Atomic Type Specifiers** from the Common Lisp specification [Ans94, Section 4.2.3 Type Specifiers].

```
(arithmetic-error                   function            simple-condition
 array                              generic-function    simple-error
 atom                               hash-table          simple-string
 base-char                          integer             simple-type-error
 base-string                        keyword             simple-vector
 bignum                             list                simple-warning
 bit                                logical-pathname    single-float
 bit-vector                         long-float          standard-char
 broadcast-stream                   method              standard-class
 built-in-class                     method-combination  standard-generic-function
 cell-error                         nil                 standard-method
 character                          null                standard-object
 class                              number              storage-condition
 compiled-function                  package             stream
 complex                            package-error       stream-error
 concatenated-stream                parse-error         string
 condition                          pathname            string-stream
 cons                               print-not-readable  structure-class
 control-error                      program-error       structure-object
 division-by-zero                   random-state        style-warning
 double-float                       ratio               symbol
 echo-stream                        rational            synonym-stream
 end-of-file                        reader-error        t
 error                              readtable           two-way-stream
 extended-char                      real                type-error
 file-error                         restart             unbound-slot
 file-stream                        sequence            unbound-variable
 fixnum                             serious-condition   undefined-function
 float                              short-float         unsigned-byte
 floating-point-inexact             signed-byte         vector
 floating-point-invalid-operation   simple-array        warning
 floating-point-overflow            simple-base-string
 floating-point-underflow           simple-bit-vector    )
```

10.2.9 Pool: Intersections and Unions

This pool contains the type specifiers which are `(and ...)` and `(or ...)` combinations of the types in Section 10.2.8. Starting from this list, we randomly generated type specifiers using `and` and `or` combinations of names from this list such as the following:

```
(arithmetic-error function array sequence
  (and arithmetic-error function)  (or arithmetic-error function)
  (or function array)  (or function sequence)
  ...)
```

10.2.10 Pool: Subtypes of `fixnum` using `member`

This pool contains type specifiers of various subtypes of `fixnum` all of the same form, using the `(member ...)` syntax.

```
((member 2 6 7 9) (member 0 2 7 8 9) (member 0 2 5 6) (member 2 4 5)
 (member 0 1 2 4 6 8 10) (member 0 2 3 4 5 6 8 9) (member 1 2 3 4 5 6 10)
 (member 3 5 6 7 8) (member 0 1 3 5 8 9) (member 1 2 4 5 8 10)
 (member 0 2 5 6 8 9 10) (member 0 2 3 4) (member 1 4 5 6 7 9 10)
 (member 0 2 3 4 5 7 9) (member 3 4 5 9) (member 1 3 6 7 8 9 10)
 (member 0 1 2 3 6 10) (member 0 1 3 4 5 6 9 10) (member 1 2 8 10)
 (member 1 3 6 7 8 10) (member 0 1 2 4 10) (member 0 1 2 3 4 6 7 9)
 (member 0 1 2 4 5 6 7 8 10) (member 0 4 5 7 9 10) (member 1 3 4 6 9))
```

10.3 MDTD algorithm implementations

In the following sections we analyze the performance results of several functions which implement the MDTD algorithm. Here is a summary of the functions by name.

Function name	Algorithm	Data structure	Hash strategy	`subtypep`
`mdtd-baseline`	baseline	s-expr		CL
`mdtd-bdd-strong`	baseline	BDD	strong	CL
`mdtd-bdd-weak`	baseline	BDD	weak	CL
`mdtd-bdd`	baseline	BDD	dynamic	CL
`mdtd-rtev2`	Algorithm 7	s-expr		CL
`mdtd-graph`	graph	s-expr		CL
`mdtd-graph-baker`	graph	s-expr		Baker
`mdtd-bdd-graph-weak`	graph	BDD	weak	CL
`mdtd-bdd-graph`	graph	BDD	dynamic	CL
`parameterized-mdtd-bdd-graph`	graph	BDD	dynamic	CL
`mdtd-bdd-graph-strong`	graph	BDD	strong	CL
`mdtd-bdd-graph-baker`	graph	BDD	dynamic	Baker
`mdtd-sat`	SAT	s-expr		CL

Figure 10.2: Various MDTD functions

10.4 Tuning the BDD hash mechanism

Section 5.3.3 presented a very basic procedure for establishing a dynamically scoped hash table which is used by `bdd` object allocation. That section mentions that there are alternative implementations of `bdd-call-with-new-hash` which can affect run-time performance. In this section (Section 10.4) we look at some possible implementations and compare the performance of the MDTD algorithm on the pools described in Section 10.2.

As mentioned in Section 5.3, a challenge posed by the Bryant/Brace [Bry86, BRB90] optimization is that the size of hash table may become many orders of magnitude larger than the number of `bdd` nodes actually used at any one time. The probability of a node in the hash table ever being reused is exceedingly small. The number of nodes in an n-variable ROBDD is less than 2^{n+1}, but the number of Boolean functions of n variables is 2^{2^n}. So the probability of reuse for large ROBDDs asymptotically approaches

$$\frac{2^{n+1}}{2^{2^n}} = 2^{1+n-2^{n-1}}.$$

One way to optimize `bdd-call-with-new-hash` in Implementation 5.18 is to arrange that nodes which are no longer referenced elsewhere, be subject to garbage collection. This would mean that, if the table gets large, then nodes which are currently being referenced should remain in the table so as to enforce the merging rule; however, other nodes should be expunged.

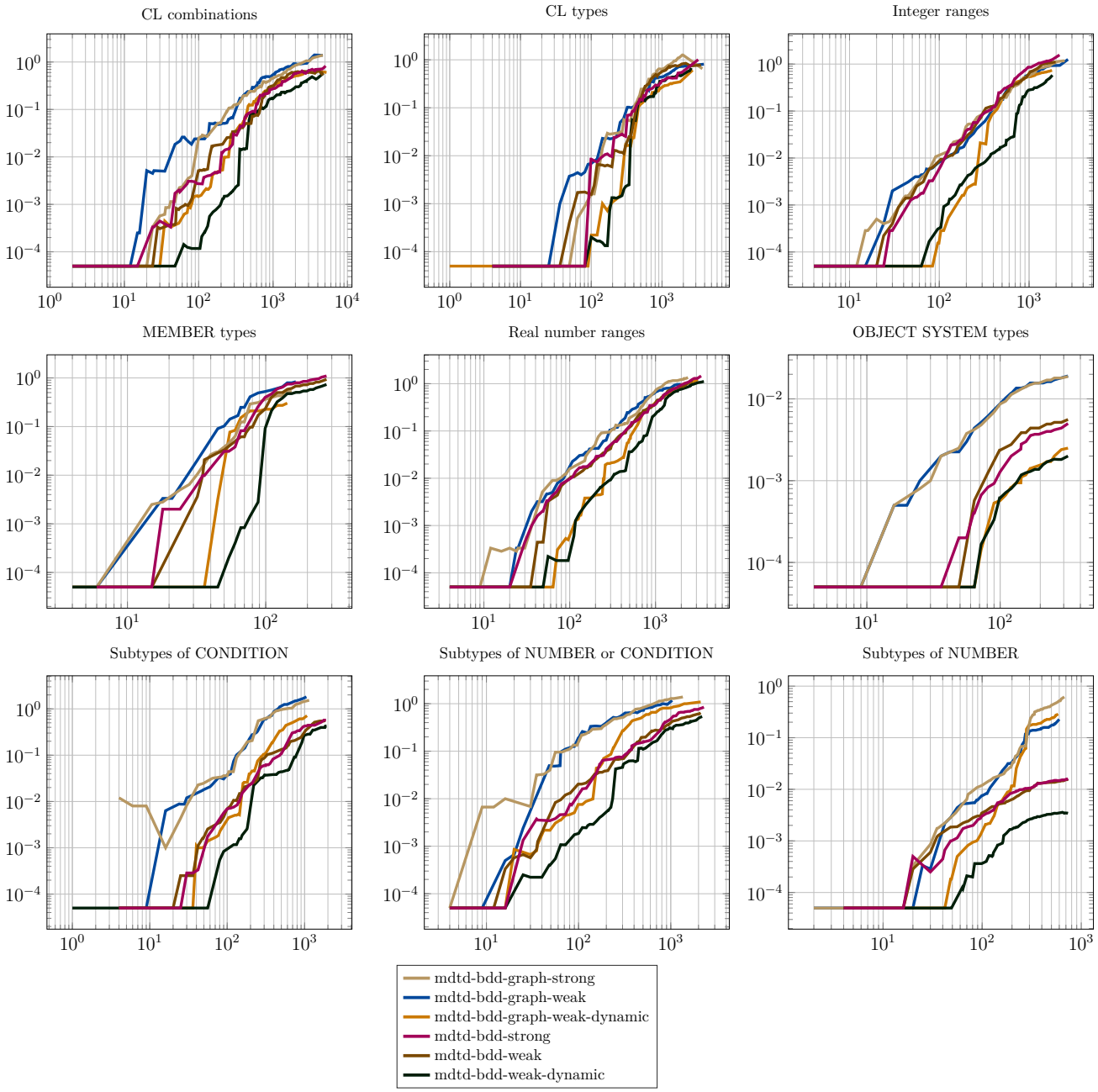
Some Common Lisp implementations, notably SBCL, provide such a hash table, called a `weak` hash table. By default, a hash table is `strong`, meaning the garbage collector is inhibited from discarding any of its entries. However, a call to `make-hash-table` with the arguments `:weakness :value` creates a hash table which allows the garbage collector this special privilege. Implementation 10.5 shows three functions defined as: `bdd-call-with-new-hash-strong`, `bdd-call-with-new-hash-weak` and `bdd-call-with-new-hash-weak-dynamic`. The function `bdd-call-with-new-hash-strong` is the same as previously seen in Implementation 5.18, *i.e.* with a strong hash table. The function `bdd-call-with-new-hash-weak` implements the weak hash table. The function `bdd-call-with-new-hash-weak-dynamic` is a bit different from `bdd-call-with-new-hash-weak` in recursive calls. If during the dynamic extent of a call to `bdd-call-with-new-hash-weak`, the same function, `bdd-call-with-new-hash-weak` is called again, a new hash table is allocated and consequently, new `bdd` nodes are sought and allocated in this one. However, during the entire dynamic extent of `bdd-call-with-new-hash-weak-dynamic`, any successive call to `bdd-call-with-new-hash-weak-dynamic` does not allocate any new hash table. This *weak-dynamic* capability allows several related ROBDD computations to be done using the same hash table, effectively implementing a forest of `bdd` nodes–many `bdd` top nodes which share subtrees between them.

Implementation 10.5 (Alternative implementations of `bdd-call-with-new-hash`)**.**

```
(defun bdd-call-with-new-hash-strong (thunk)
  (let ((*bdd-hash* (make-hash-table :test #'equal)))
    (funcall thunk)))

(defun bdd-call-with-new-hash-weak (thunk)
  (let ((*bdd-hash* (make-hash-table :test #'equal
                                     :weakness :value)))
    (funcall thunk)))

(defun bdd-call-with-new-hash-weak-dynamic (thunk)
  (if *bdd-hash*
      (funcall thunk)
      (bdd-call-with-new-hash-weak thunk)))

(defvar *bdd-call-with-new-hash*  #'bdd-call-with-new-hash-weak-dynamic)

(defun bdd-call-with-new-hash (thunk)
  (funcall *bdd-call-with-new-hash* thunk))
```

In the experiment, we tested several functions on the same pools of type specifiers. Descripions of these functions can be found in Figure 10.2. The names of the functions indicate which strategy was used for `bdd-call-with-new-hash`, and can also be found in the **Hash strategy** column of Figure 10.2.

The results are shown in Figure 10.3 were it can be seen that the best performing functions were those using the *weak dynamic* approach: `mdtd-bdd-weak-dynamic` and `mdtd-bdd-graph-weak-dynamic`.



Function name	Algorithm	Hash strategy
`mdtd-bdd-strong`	baseline	strong
`mdtd-bdd-weak`	baseline	weak
`mdtd-bdd-weak-dynamic`	baseline	weak dynamic
`mdtd-bdd-graph-strong`	graph	strong
`mdtd-bdd-graph-weak`	graph	weak
`mdtd-bdd-graph-weak-dynamic`	graph	weak dynamic

Figure 10.3: Performance comparison using different hash table strengths. Each plot is displayed with y='Computation Time (seconds)' *vs.* x='Product Size'.

10.5 Tuning the BDD-based graph algorithm

Section 9.4 explains how we represent the problem of set decomposition as a graph problem. That section explains how to construct the graph, and several deconstruction operations. What is, however, not explained is how to deconstruct the graph efficiently in terms of execution time. In the current section we explain the process we used in determining a reasonable way to combine the pieces into a coherent algorithm which performs well

on diverse pools. We do not claim our approach is ideal, but it does attempt to take many different concerns into account and it provides a way to do a visual sanity-check of the final results.

From the abstract, high-level view, once the graph (such as Figure 9.4) has been constructed, the algorithm proceeds by visiting each node and applying some of the set of possible operations. Section 9.4 does not make any claim about which order of operations is better in terms of execution speed, other than to offer two algorithms as alternatives: Algorithm 8 and Algorithm 9. The motivation for the need of alternative algorithms is that the subset extraction operation has multiple variations, two of which are shown in Algorithm 16 (Section 9.4.2) and Algorithm 17 (Section 9.4.3). Other variations are outlined here.

Break subset:	`strict` *vs.* `relaxed`, explained in Sections 9.4.2 and 9.4.3.
Break loop:	`yes` *vs.* `no`, explained in Section 9.4.5.
Iteration topology:	$\forall$ node $\forall$ operation *vs.* $\forall$ operation $\forall$ node, explained in Section 9.4.7.
Recursive:	`yes` *vs.* `no`, explained in Section 9.4.6 and 9.4.7
Node visitation order:	`"SHUFFLE"`, `"INCREASING-CONNECTIONS"`, `"DECREASING-CONNECTIONS"`, `"BOTTOM-TO-TOP"`, `"TOP-TO-BOTTOM"`, explained in Section 9.4.7.
Inclusion:	`implicit` *vs.* `explicit`, explained in Section 9.4.1.
Empty set:	`late-check`, *vs.* `correct-label` *vs.* `correct-connection`, explained in Section 9.4.6.

The natural questions which arise are: what is the best way to assemble these pieces into an efficient algorithm, and which of the variations of each operation are the best to use? In the current section we describe some experimentation we did to determine a reasonable approach for a wide range of pools.

Section 10.2 details 10 pools which were constructed to test various aspects of the MDTD algorithm. We have constructed test suites which run on each of the data pools testing the performance of the graph-based algorithm as a function of each of the parameters described above, with the exception of *inclusion* and *empty-set*. At this point in our experimentation we have not yet included these parameter in our performance testing.

Some of the parameter options don't make sense together. For example, we cannot use *break subset=relaxed* along with *break loop=no* because this would result in some graphs which cannot be reduced at all. Of the remaining possible combinations, we have constructed 45 different variations of the input parameters. Figure 10.6 shows the performance results in terms of calculation time *vs.* product size. The red curves indicate the default performance of the algorithm (having been tuned by the analysis explained below). The black curves indicate the minimum time (best possible) performance for each data pool. In the case that only red or only black is visible, this means the two curves exactly coincide. (The following paragraphs explain what we mean by *best performance*). The plots show that although the default parameters do not yield the best results on any one of the pools, the results do seem reasonably good.

For each of the 10 test pools, 45 curves were plotted. In each case we wanted to determine the *fitness* of the 45 curves. There are potentially many different ways of judging the curves for *fitness*.

As all of our sampled data is positive (calculation time is always non-negative), we elected to derive a *norm* for a curve. *I.e.*, we can sort the curves from best to worst by order of increasing value of the norm. A small norm means fast calculation time over the range sampled, and large norm means slow calculation time. But even then, there are many ways of calculating a norm of sampled functions. A simple Google search of the terms "metrics for sampled functions" found 726,000 results of scholarly papers.

Three possible norms were considered:

- $\|f\|_\mu$, Average value, *i.e.*, sum of y-values divided by number of samples. Definition 10.6. This is the norm we decided to use in our experiments.
- $\|f\|_1$, Average value based on numerical integral divided by size of domain. Definition 10.7.
- $\|f\|_{rms}$, RMS (root means square) distance from point-wise minimum over all the curves. Definition 10.8.

Definition 10.6. Suppose the curve $\mathscr{C}$ is the set of x-y pairs comprising a curve, and that $|\mathscr{C}|$ denotes the number of points in the set. We define

$$\|f\| = \frac{1}{|\mathscr{C}|} \sum_{(x,y)\in\mathscr{C}} y \,.$$

Definition 10.7. Suppose that $[(x_0, y_0), (x_1, y_1)...(x_m, y_m)]$ is the vector of x-y pairs comprising the curve. We define

$$\|f\|_1 = \frac{1}{x_m - x_0} \sum_{i=1}^{m} \frac{y_i + y_{i-1}}{2} \cdot (x_i - x_{i-1}) .$$

Definition 10.8. Denote the set of all 45 sampled functions for this pool as $\{f_j\}_{j=1}^{45}$, such that f_j is interpolated linearly between the sampled points. Denote the set of all sampled x-values as $\{x_i\}_{i=0}^{M}\}$ with $x_0 < x_1 < ... < x_M$. Denote the point-wise minimum function as

$$\check{f} \colon \{x_i\}_{i=0}^{M} \to \mathbb{R} \mid \check{f}(x_i) = \min \left\{ f_j(x_i) \right\}_{j=1}^{45}$$

Now we define

$$\|f\|_{rms} = \frac{1}{x_M - x_0} \sqrt{\sum_{i=2}^{45} \left[f(x_i) - \check{f}(x_i) \right]^2 \cdot (x_i - x_{i-1})} .$$

We decided that the most reasonable of these for our needs is the $\|f\|_\mu$ average. The other two norms were dismissed because they don't give equal weighting to all the samples. We found that the latter of the two norms would give excessive weighting to large values of calculation time *vs.* sample size.

Definition 10.9. If μ is the average norm for a pool and σ is the standard deviation of the norms, then the Student score [Gos08], Z, (sometimes called Z-score) is defined as:

$$Z(f) = \frac{||f|| - \mu}{\sigma} .$$

More information on this curious name, Student score, may be found in Section 10.11.

Once the norm of each curve is calculated, (45 for each pool, over 10 pools), we calculate a *Student score* for each curve in the same pool. The smallest (most negative) Z indicates the *best* curve, *i.e.*, the set of parameter values resulting in the least calculation time within the pool. Using the Student score allows us to compare results from different pools, as each score has already been *normalized* by the standard deviation of its respective pool.

Once each curve had been scored (via the Student score), we proceeded to determine which sets of parameters lead to the best Student score. We considered, one by one, each of the parameters, and for each possible value of the parameter, we calculated the average Student score (across all 10 pools) of all the curves which use that parameter value.

Parameter	**value**
Break subset	relaxed
Break loop	no
Iteration topology	operations per node
Recursive	yes
Node visitation order	BOTTOM-TO-TOP
Inclusion	not tested
Empty set	not tested

Figure 10.4: Experimentally determined best parameter values.

Example 10.10 (Experiment to determine best value of *break subset*)**.**
There are two possible values of *break subset*: *strict* and *relaxed*. We collected the curves of all the functions

for which *break subset = strict*, and all the curves for which *break subset = relaxed*, and averaged the Student scores of the two sets. The average Student score, as seen in Figure 10.5, for *break subset=strict* was 0.21279174 and for *break sub=relaxed* was -0.06591506. From this, we infer that *relaxed* is a better choice than *strict*. Figure 10.4 shows the experimentally determined best values for the parameters of the graph based algorithm.

Ranking Results		
Parameter	Value	Average Student Score
do-break-loop	nil	-0.101393685
	t	0.05745659
do-break-sub	relaxed	-0.06591506
	strict	0.21279174
inner-loop	operation	-0.043741602
	node	0.1684451
recursive	t	-0.14755104
	nil	0.06916463
sort-strategy	BOTTOM-TO-TOP	-0.24243449
	TOP-TO-BOTTOM	-0.21874332
	DECREASING-CONNECTIONS	-0.14941746
	INCREASING-CONNECTIONS	-0.118050456
	SHUFFLE	0.86358154

Figure 10.5: Summary of results from Figure 10.6. For each parameter, we display each possible value, and the corresponding average Student score. The minimum (most negative) Student score indicates the best value for the parameter in question.

Given this experimentally determined set of globally best values for the parameters, we have candidates for optimum default values of those parameters. Therewith, we define the default function. These experimentally tuned values are shown in Figure 10.5. The performance of this default function relative to the other possible parameterizations can be seen in Figure 10.6. Each figure shows the performance of the default function (in red) and the performance of the per-pool best function (in black) along with the 43 other functions for that pool. As a visual sanity check of the plots, we observe that although the default parameters do not yield the best results on all the pools, the results do seem very good; they coincide with the best curve in several of the pools and are visibly close the best curves for the other pools.

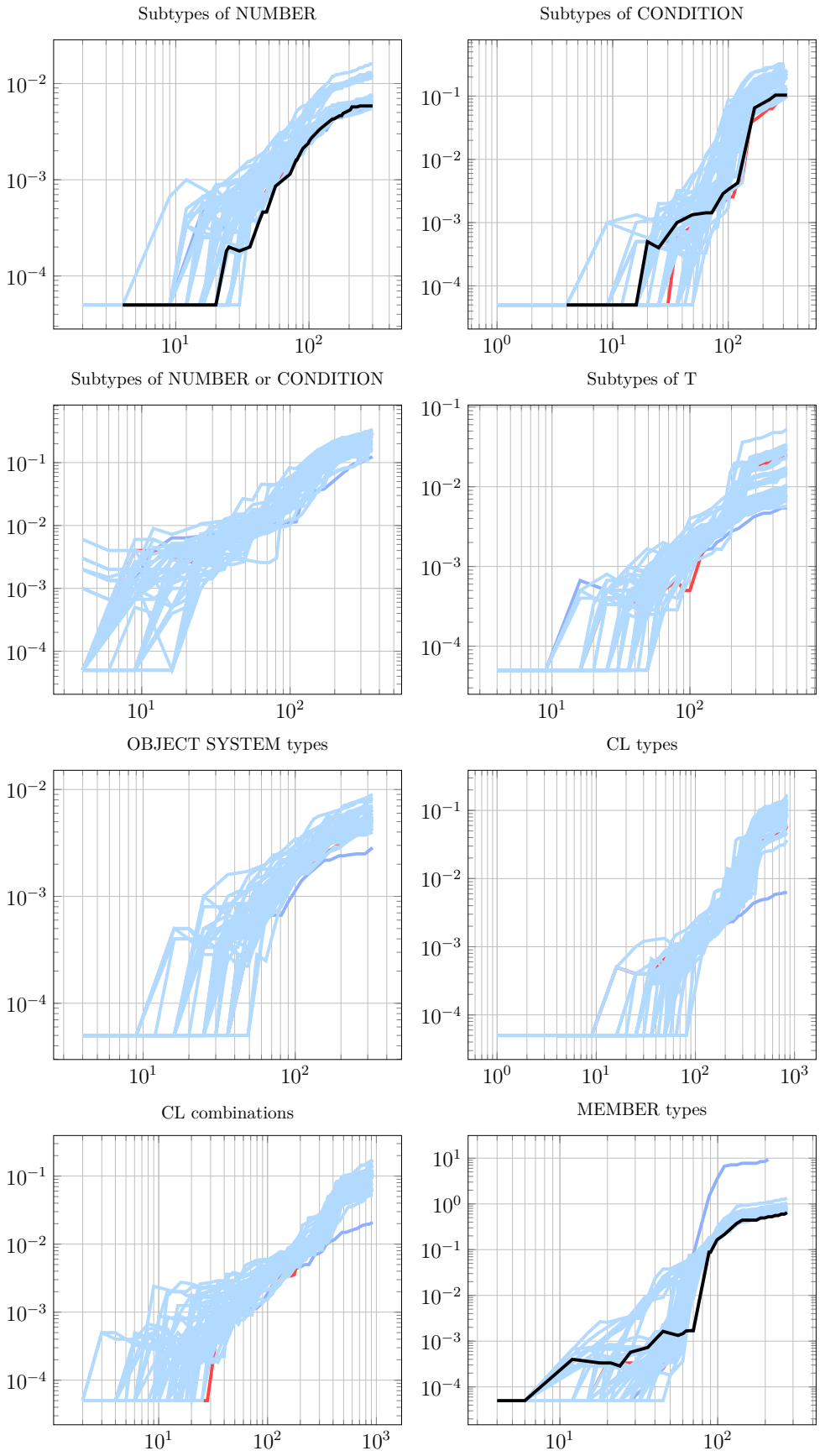


Figure 10.6: Tuning graph-based algorithm with various pools. Each plot is displayed with y='Computation Time (seconds)' *vs.* x='Product Size'.

10.6 Analysis of performance tests

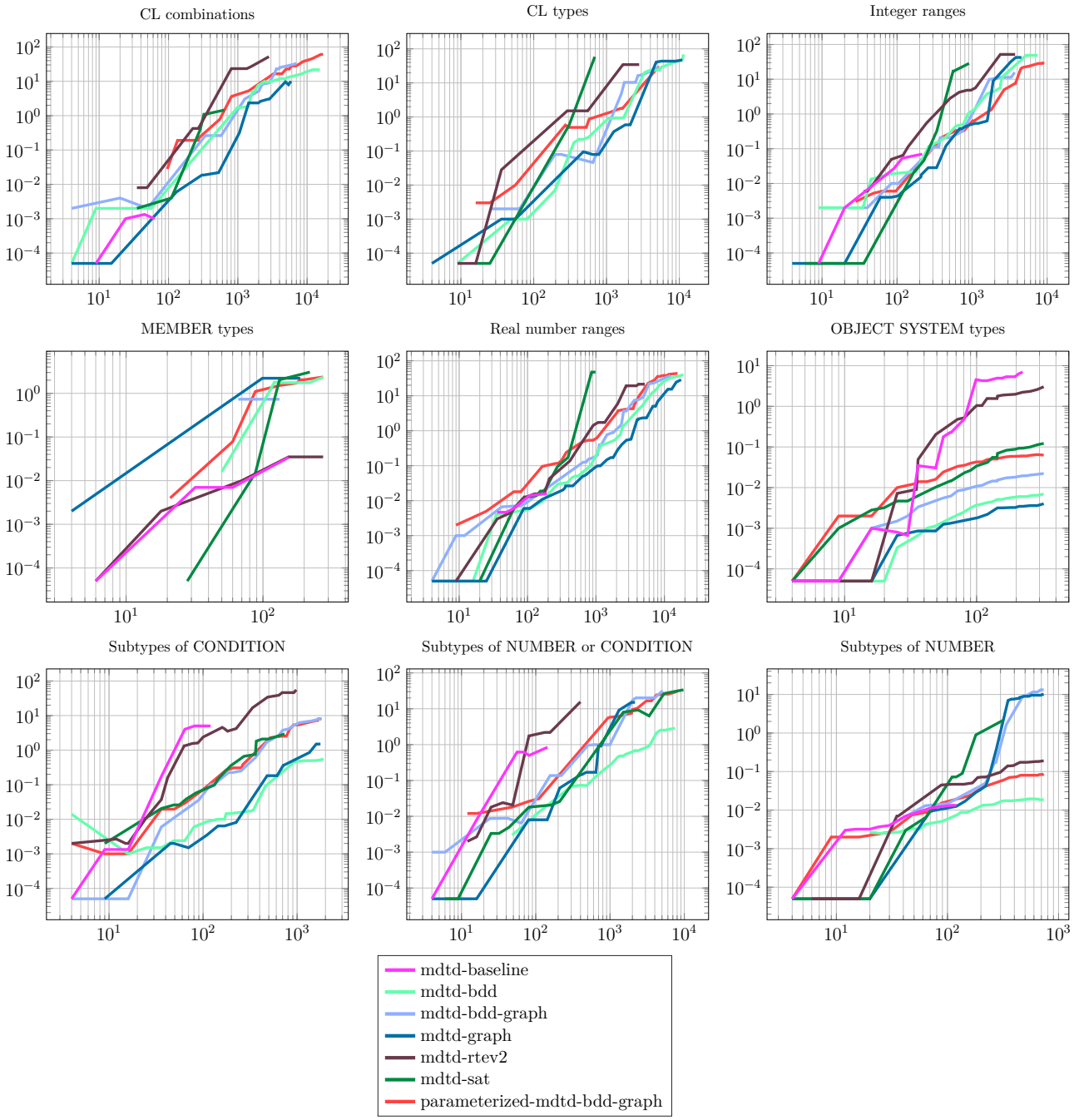
After tuning the hash table mechanism (Section 10.4) and the graph-based MDTD algorithm parameters (Section 10.5), we were able to test several different algorithms against each other.

Figure 10.7 contains several graphs showing the performance of the functions on different pools. We find several of the results surprising or even strange:

- No single one of the algorithms performs best in most cases. Even the algorithms which we guessed would be slow, perform best in some cases.
- The function `mdtd-sat` does very well for small product size, even thought it has exponentially complexity.
- The baseline algorithm, `mdtd-baseline`, does very well with the MEMBER pool.
- The function `mdtd-rtev2`, which is supposed to be an improving over `mdtd-baseline` often performs worse, such as in the CL combinations pool, CL types pool, and in the NUMBER pool for computations exceeding 0.1 seconds.

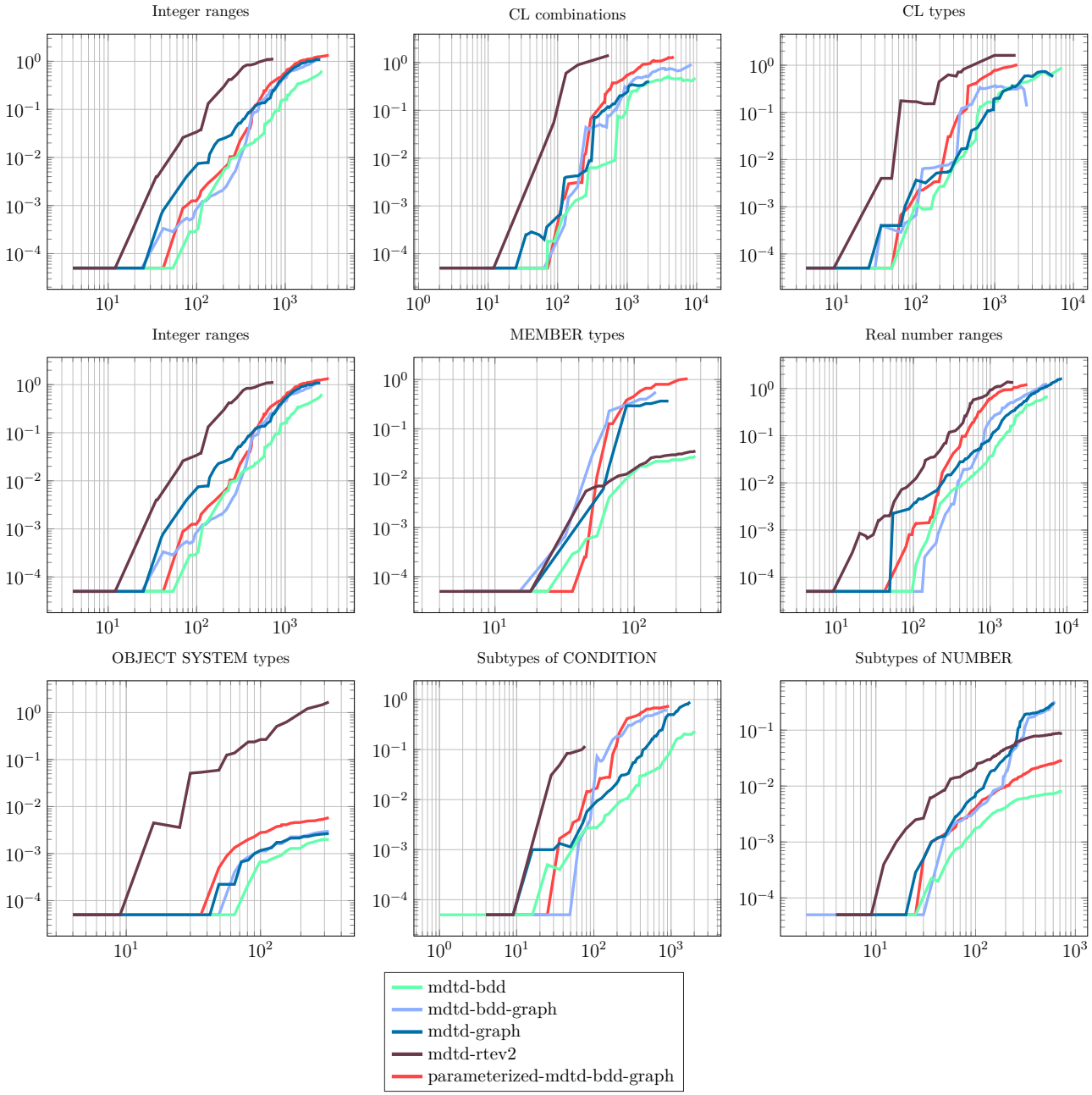
There is no clear winner for small sample sizes. But it seems the tree based algorithms do very well on large sample sizes. This is not surprising, as the graph-based algorithm was designed with the intent to reduce the number of passes and take advantage of subtype and disjointness information.

Figure 10.8 contains several graphs contrasting some of the effective algorithms in terms of execution time *vs.* sample size. The product size is the integer product of the number of input types multiplied by the number of output types. This axis choice was chosen as it heuristically seems to be the best to demonstrate the relative performance for the different pools. Each of the graphs shows how the same algorithms perform on different pools, one pool per plot.



Function name	**Algorithm**	**Data structure**	**Hash strategy**	**subtypep**
`mdtd-baseline`	baseline	s-expr		CL
`mdtd-bdd`	baseline	BDD	dynamic	CL
`mdtd-graph`	graph	s-expr		CL
`mdtd-bdd-graph`	graph	BDD	dynamic	CL
`parameterized-mdtd-bdd-graph`	graph	BDD	dynamic	CL
`mdtd-rtev2`	Algorithm 7	s-expr		CL
`mdtd-sat`	SAT	s-expr		CL

Figure 10.7: Performance comparison over several pools using different algorithms. Each plot is displayed with y='Computation Time (seconds)' *vs.* x='Product Size'.



Function name	Algorithm	Data structure	Hash strategy	`subtypep`
`mdtd-bdd`	baseline	BDD	dynamic	CL
`mdtd-bdd-graph`	graph	BDD	dynamic	CL
`mdtd-graph`	graph	s-expr		CL
`mdtd-rtev2`	Algorithm 7	s-expr		CL
`parameterized-mdtd-bdd-graph`	graph	BDD	dynamic	CL

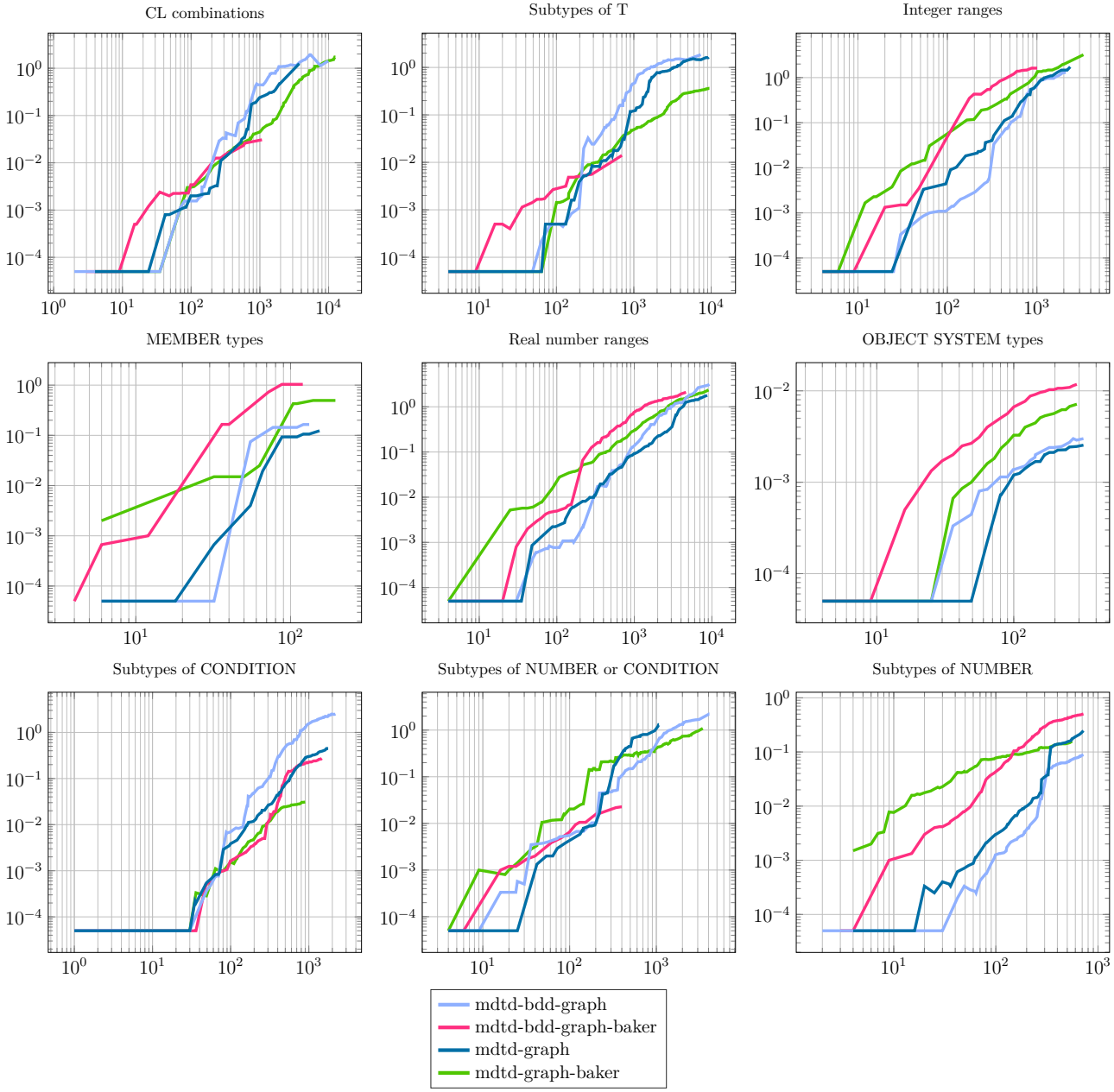
Figure 10.8: Performance comparison over several pools using best five algorithms. Each plot is displayed with y='Computation Time (seconds)' *vs.* x='Product Size'.

10.7 Analysis of performance tests with Baker functions

Section 9.6 introduces a Baker [Bak92] `subtypep` variant of the type decomposition functions. The plots shown in Figure 10.9 show the very preliminary results of performance tests of our implementations of the functions using `cl:subtypep` and `baker:subtypep`. We emphasize that these are preliminary results, because at the time of this publication, Valais [Val18] has not yet released the code publicly; it has not yet been fully optimized; and it is known to have some unresolved bugs at the time of the publication of this report. Nevertheless, we

believe the results are revealing in a couple of ways.

- In several of the pools the function `mdtd-bdd-graph-baker` is incomplete, not extending above about *two* $\times 10^{-2}$ seconds in the CL combinations, Subtypes of `T`, and Subtypes of `number` or `condition` pools. We have not yet explained why this happens.
- The Baker `subtypep` implementation of `mdtd-graph-baker` does well in several pools: CL combinations, Subtypes of `T`, Subtypes of `condition`, and Subtypes of `number` or `condition`. In these pools the Baker based s-expression based graph algorithm outpaces the BDD counterparts, especially for large product sizes.



Function name	**Data structure**	**`subtypep`**
`mdtd-bdd-graph`	BDD	`cl:subtypep`
`mdtd-graph`	s-expr	`cl:subtypep`
`mdtd-bdd-graph-baker`	BDD	`baker:subtypep`
`mdtd-graph-baker`	s-expr	`baker:subtypep`

Figure 10.9: Performance comparison using different Baker algorithm Each plot is displayed with y='Computation Time (seconds)' *vs.* x='Product Size'.

10.8 Analysis of profile tests

The next series of tests were similar to those mentioned in Section 10.6. Again we made calls to various MDTD algorithms on randomly selected sets of type specifiers from the various pools (Section 10.2). However, this time, rather than simply measuring execution time, we ran each test in the SBCL deterministic profiler, to measure how much time is being spent in each function. From these measurements we created plots of percentage of time spent in each Lisp function as a function of total computation time for the call to the particular MDTD function in question. We display those plots in two different ways.

Section 10.9 displays one grid of nine plots per MDTD function, *i.e.* one plot per pool for nine of the pools. Figure 10.10 shows a nine-pool view of performance of the function `parameterized-mdtd-bdd-graph`; Figure 10.11 shows a nine-pool view of performance of the function `mdtd-bdd-graph-weak-dynamic`. Each grid of plots in Section 10.9 gives an impression of the run-time profile of one particular MDTD function.

Section 10.10 shows the same plots as Section 10.9 but grouped differently, showing one grid of plots per pool. Each plot within the grid shows the profile of one particular MDTD function. Figure 10.25, for example, shows the profiles of nine MDTD functions for the pool "Subtypes of NUMBER or CONDITION".

At the time of this publication, the SBCL deterministic profiler does not have a programmatic interface to its output. Rather than attempting to use internal APIs, we elected to parse the textual output of the profiler, which looks something like the following.

```
  seconds  |     gc     |    consed   |  calls |  sec/call  |  name
-------------------------------------------------------------------------------
     1.314 |      0.000 | 763,854,800 | 14,718 |   0.000089 | RND-ELEMENT
     0.974 |      0.967 |           0 |     10 |   0.097396 | GARBAGE-COLLECT
     0.317 |      0.000 |     293,328 |     20 |   0.015849 | RUN-PROGRAM
     0.007 |      0.000 |     360,448 |     10 |   0.000707 | CHOOSE-RANDOMLY
     0.001 |      0.000 |           0 |  2,120 |   0.000000 | FIXED-POINT
     0.000 |      0.000 |           0 |    520 |   0.000001 | CACHED-SUBTYPEP
     0.000 |      0.000 |           0 |    520 |   0.000000 | ALPHABETIZE
     0.000 |      0.000 |           0 |    840 |   0.000000 | CMP-OBJECTS
     0.000 |      0.000 |           0 |    520 |   0.000000 | REDUCE-MEMBER-TYPE
     0.000 |      0.000 |           0 |     10 |   0.000000 | BDD-RECENT-COUNT
     0.000 |      0.000 |   1,622,880 |  1,040 |   0.000000 | CACHING-CALL
     0.000 |      0.000 |           0 |  3,160 |   0.000000 | ALPHABETIZE-TYPE
     0.000 |      0.000 |           0 |    520 |   0.000000 | DISJOINT-TYPES-P
-------------------------------------------------------------------------------
     2.618 |      0.967 | 766,131,472 | 26,699 |            | Total
```

To obtain one data point in the plots shown in Sections 10.9 and 10.10, we evaluated a call to one of the MDTD functions from Figure 10.2 given a list of type specifiers from the pool, and thereafter obtained the profiler output text as shown above. The number of seconds shown in the first column was normalized by dividing by the total time shown in the final line. For example, the `rnd-element` function took 1.314 seconds of 2.618 total, therefore we generate a point $(x, y) = (\frac{1.315}{2.618}, 2.618) = (50.23\%, 2.618)$ for the curve `rnd-element`. We collected such data points for all such functions in the profile output with positive run-times, and repeated the process for many calls to the MDTD function with samples from the same pool.

The plot in the top left corner of Figure 10.10, representing the pool "CL combinations", generated by 360 calls to the `parameterized-mdtd-bdd-graph` function, including 281 calls which timed out after 150 seconds, and 79 calls which produced a type decomposition. This process consumed 10 hours of wall-time on a cluster node, with 72.1×10^{12} processor cycles, according to the SBCL `time` function.

```
Evaluation took:
  36093.120 seconds of real time
  36019.964000 seconds of total run time (25130.400000 user, 10889.564000 system)
  [ Run times consist of 442.340 seconds GC time, and 35577.624 seconds non-GC time. ]
  99.80% CPU
  1 form interpreted
  384 lambdas converted
  72,184,106,952,234 processor cycles
  997,174,600,304 bytes consed
```

Each run generated points for some of the internal functions shown in the table: `subtypep` (`subtypep-wrapper` is a thin wrapper which either calls `cl:subtypep` or `baker:subtypep`), `fixed-point`, `reduce-member-type` etc. The plots were constructed for each curve simply by sorting the collected points in

increasing order of total run-time (x-value). For visualization purposes we only show the top four most active functions per pool. This entire 10 hour (wall time) process was executed 150 times, as there are 10 pools and 15 MDTD functions tested.

The important thing we notice about the plots is that, as the computation time increases, a higher percentage of that computation time is dedicated, in most cases, to `subtypep`. There seem to be two exceptions to this trend. The first exception is that the graph-related functions, in some cases spend large percentage of their time in the functions `delete-green-line` and `add-green-line`, *i.e.*, in graph manipulation functions. The second exception seems to be that the MDTD functions using `baker:subtypep` have a very different profile–their time being mostly spent in the internals of `baker:subtypep`.

10.9 Profiler graphs of MDTD algorithms by pool

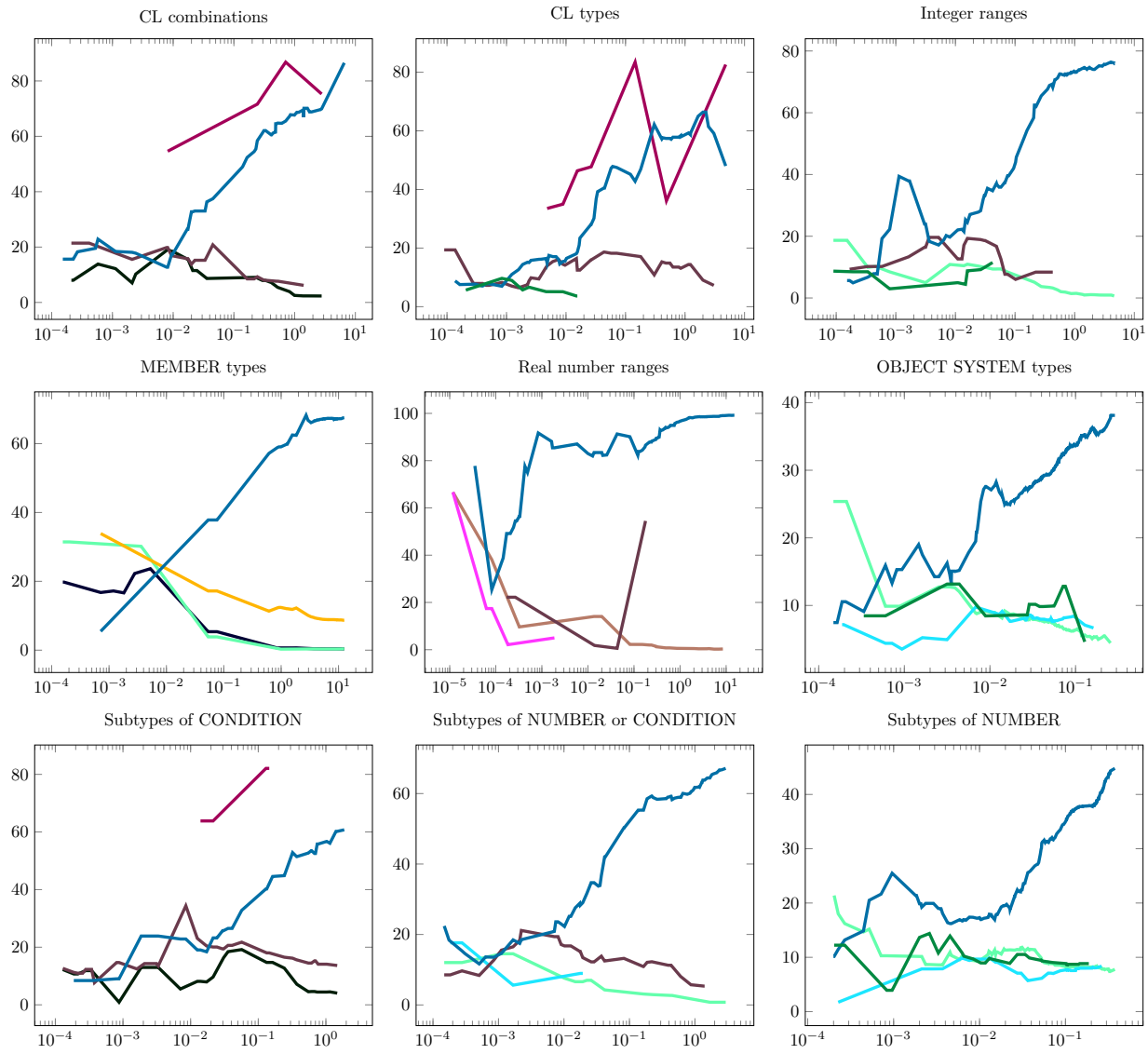


Figure 10.10: Performance Profile of various pools on algorithm **`parameterized-mdtd-bdd-graph`**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'

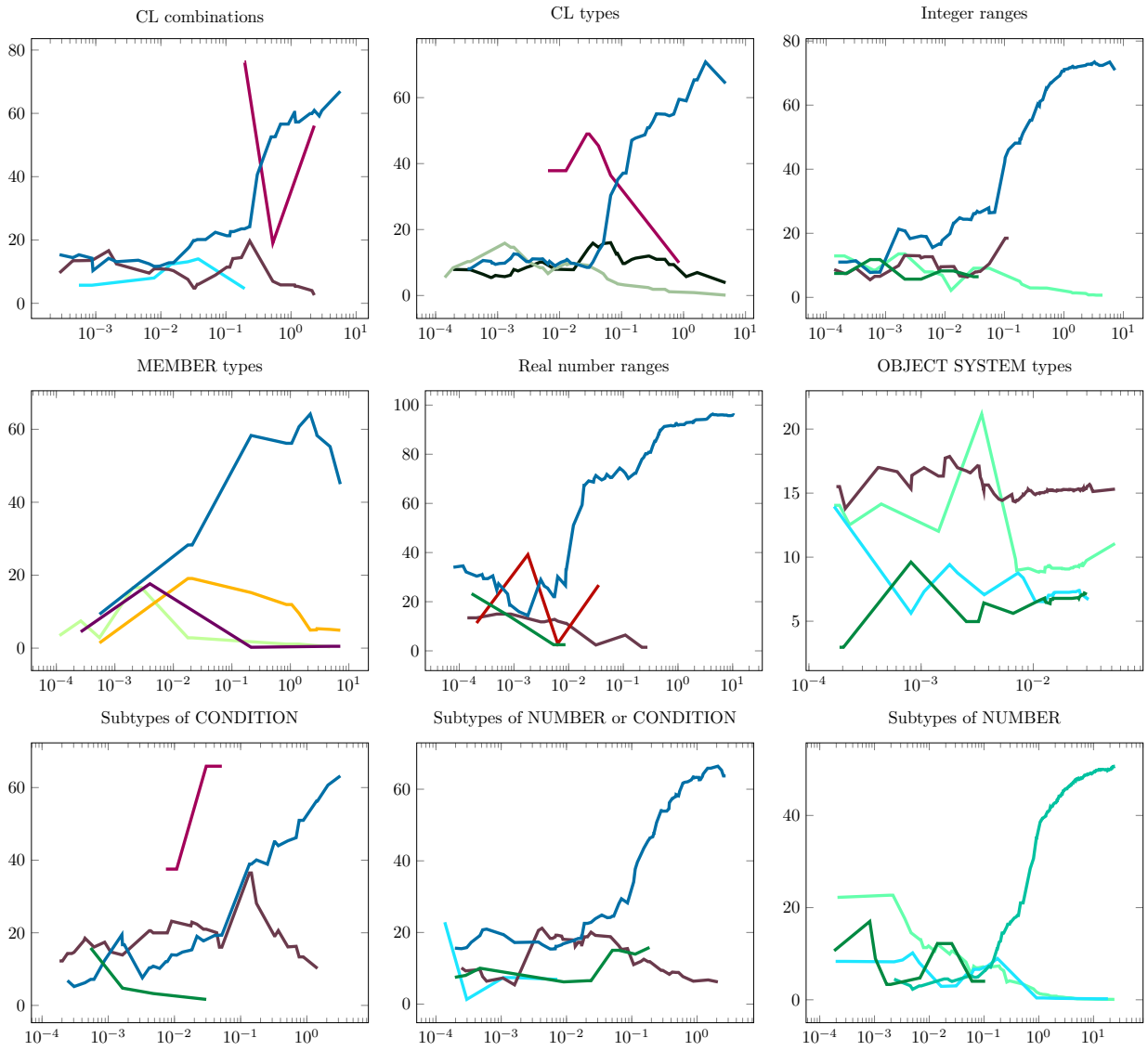
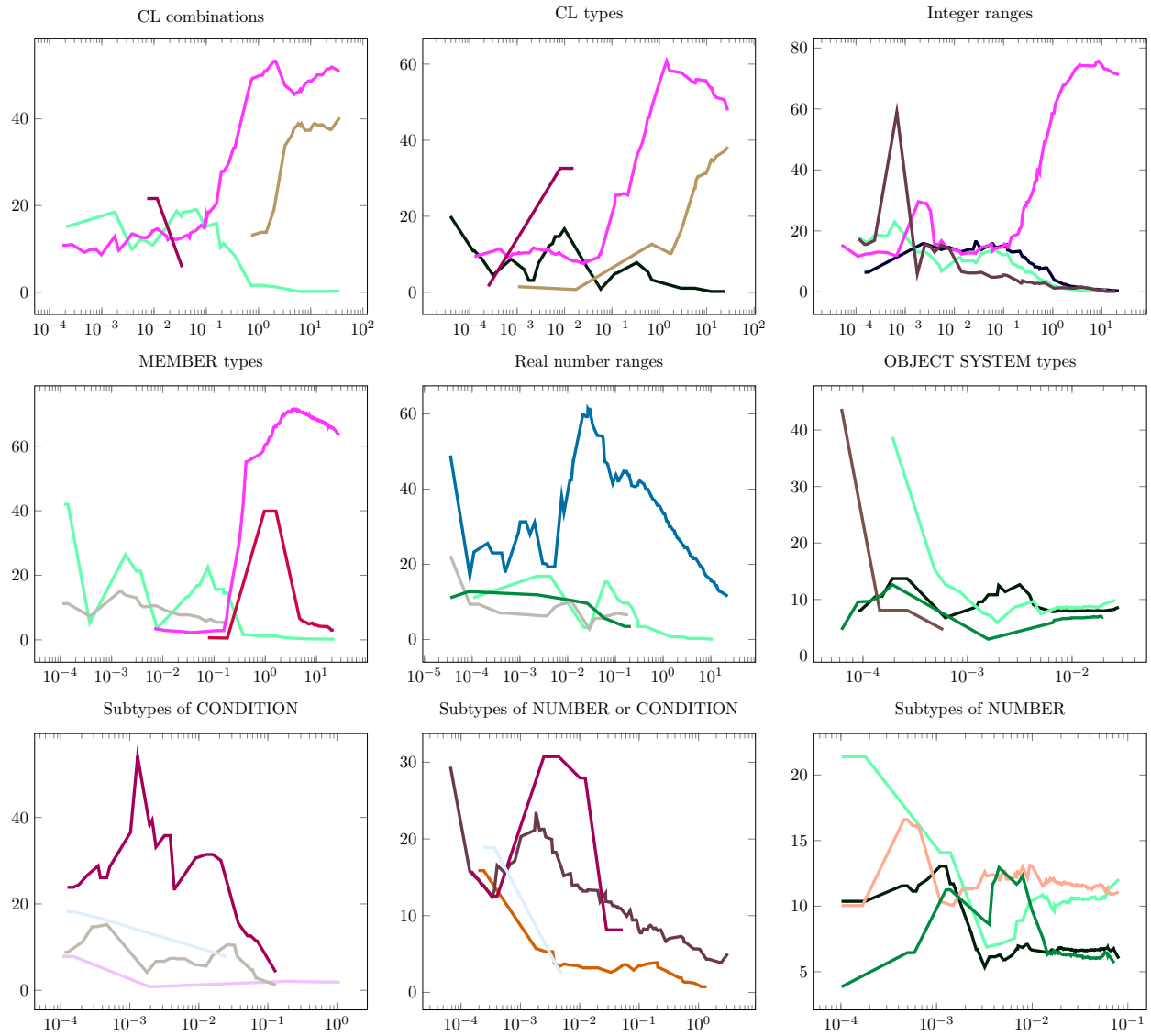


Figure 10.11: Performance Profile of various pools on algorithm **`mdtd-bdd-graph-weak-dynamic`**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'



cached-subtypep-caching-call	3342.67 seconds	245,140,975 calls
subtypep-wrapper	256.06 seconds	74,825,642 calls
smarter-subtypep-caching-call	231.86 seconds	11,203,966 calls
bdd-find	182.64 seconds	98,612,308 calls
alphabetize-type	163.27 seconds	277,943,114 calls
fixed-point	130.82 seconds	97,174,155 calls
bdd-find-int-int	125.61 seconds	67,442,808 calls
slow-disjoint-types-p	118.71 seconds	7,281,882 calls
bdd-subtypep	52.27 seconds	48,236,919 calls
type-to-dnf-bottom-up	45.38 seconds	82,175,121 calls
type-to-dnf	17.44 seconds	4,903,808 calls
bdd-find-reduction	16.40 seconds	482,046 calls
bdd-cmp	12.65 seconds	26,446,345 calls
bdd-empty-type	7.51 seconds	3,989,956 calls
bdd-and-not	6.66 seconds	2,818,048 calls
bdd	1.89 seconds	536,666 calls

Figure 10.12: Performance Profile of various pools on algorithm **mdtd-bdd-weak-dynamic**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'

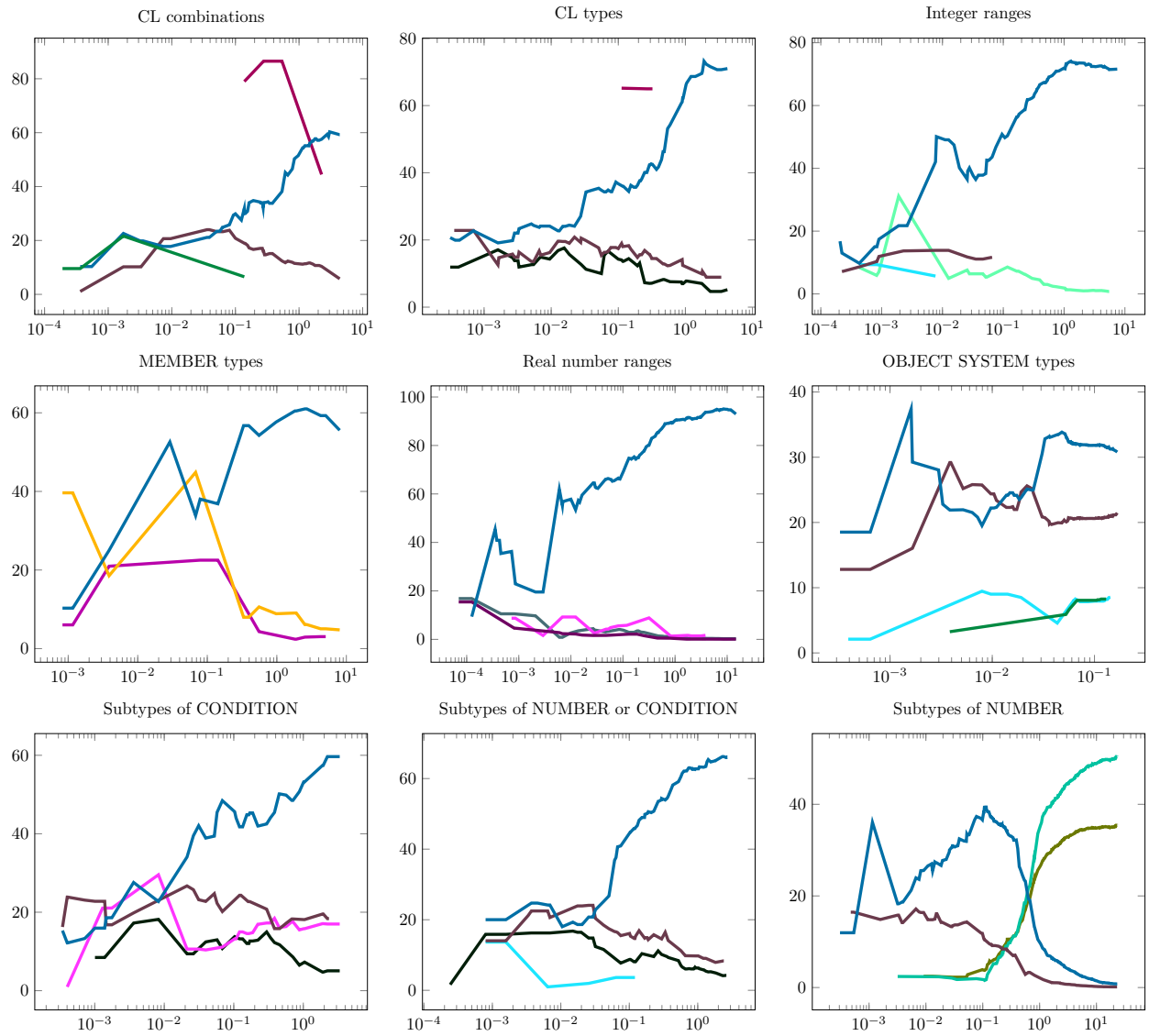


Figure 10.13: Performance Profile of various pools on algorithm **`mdtd-bdd-graph-strong`**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'

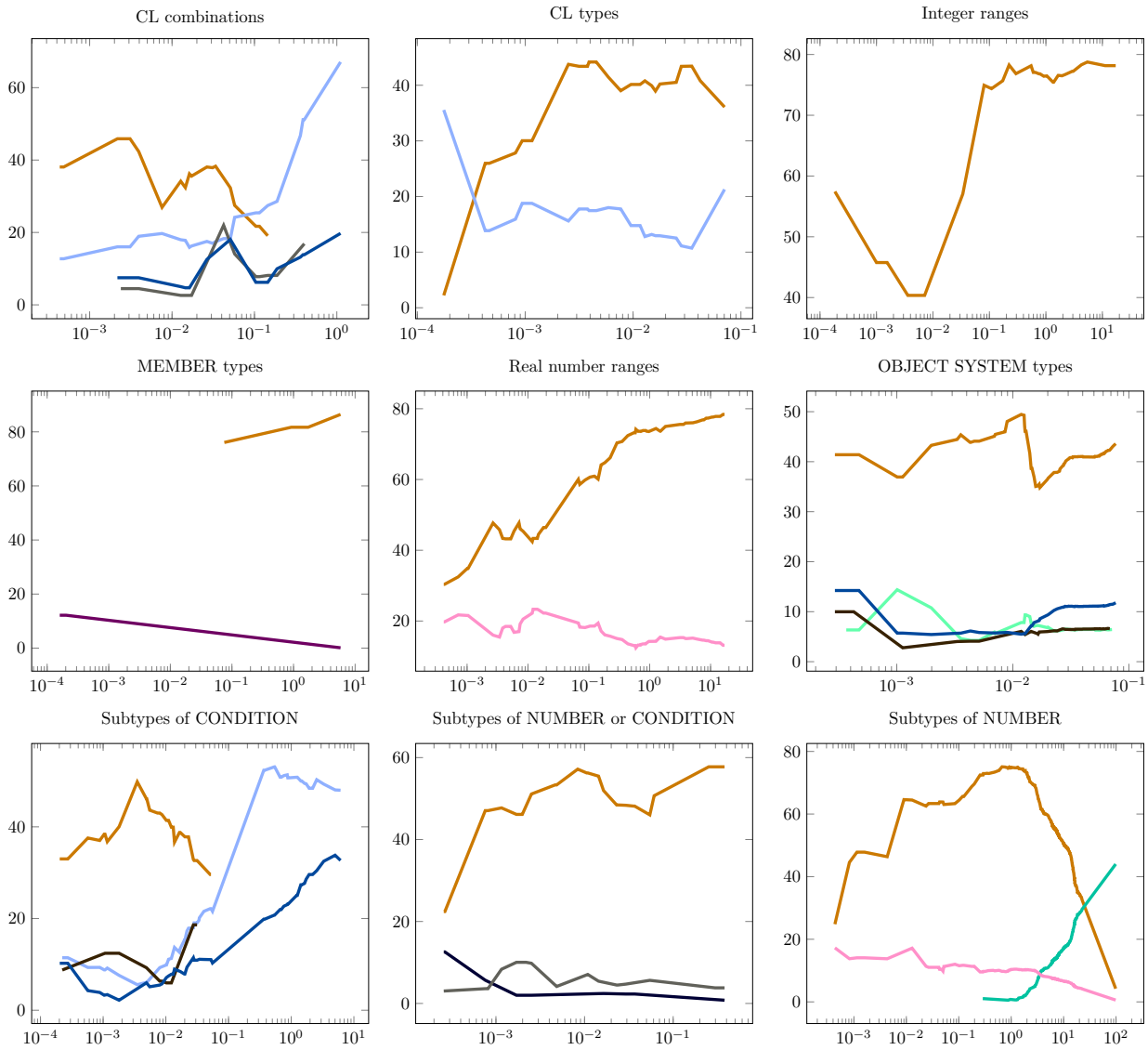
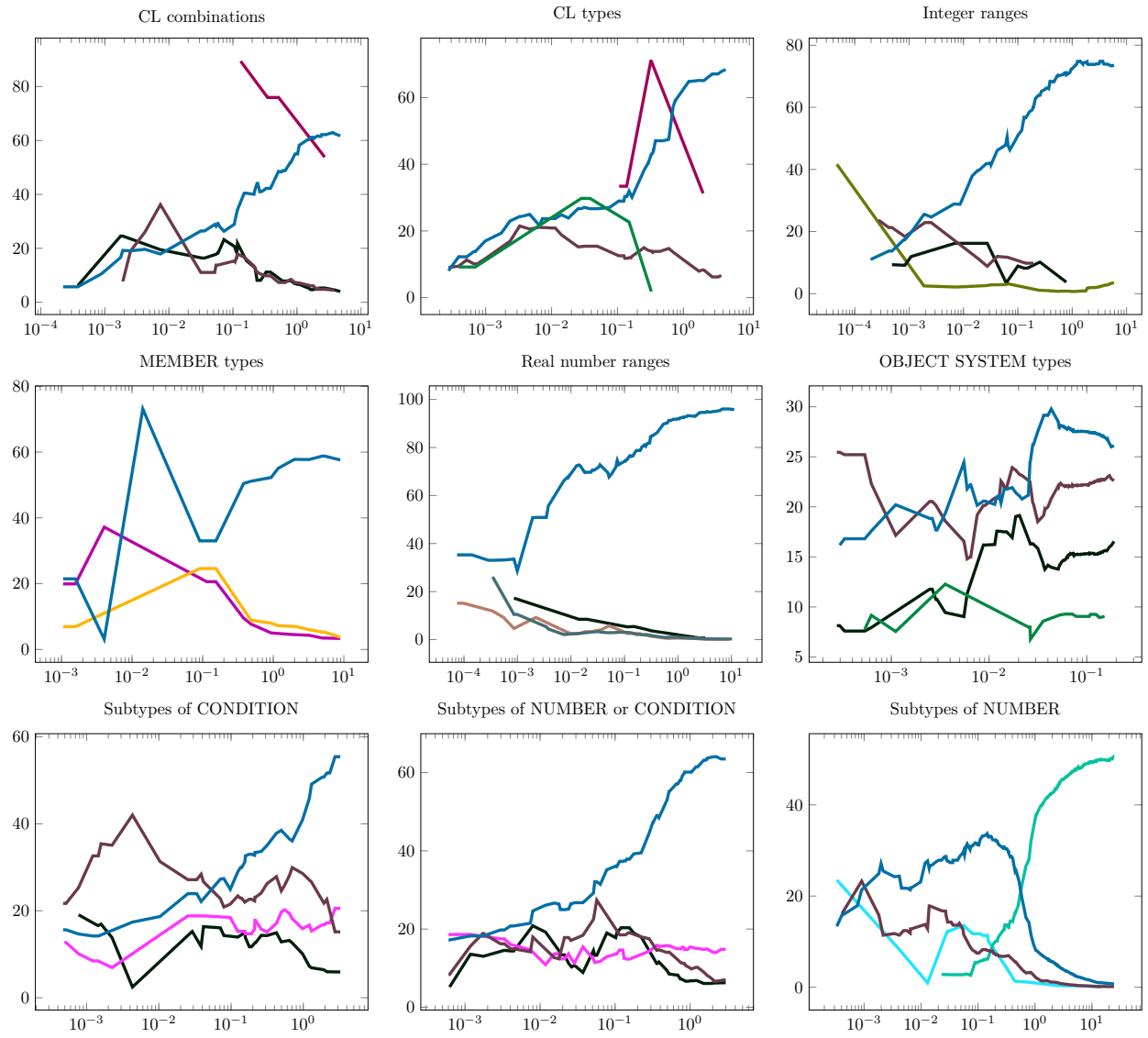


Figure 10.14: Performance Profile of various pools on algorithm **`mdtd-bdd-graph-baker`**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'



delete-green-line	1134.36 seconds	3,914,784 calls
subtypep-wrapper	925.99 seconds	592,353,496 calls
fixed-point	289.94 seconds	420,095,926 calls
alphabetize-type	107.40 seconds	379,063,296 calls
cached-subtypep-caching-call	75.82 seconds	126,705,136 calls
type-to-dnf-bottom-up	14.68 seconds	64,425,113 calls
bdd-positive	10.24 seconds	8,859,825 calls
bdd-make-key	9.70 seconds	13,416,275 calls
alphabetize	8.74 seconds	10,346,733 calls
cached-subtypep	6.02 seconds	5,227,764 calls
add-green-line	3.84 seconds	127,087 calls
slow-disjoint-types-p	3.39 seconds	107,584 calls
cmp-objects	3.04 seconds	28,235,401 calls

Figure 10.15: Performance Profile of various pools on algorithm **mdtd-bdd-graph-weak**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'

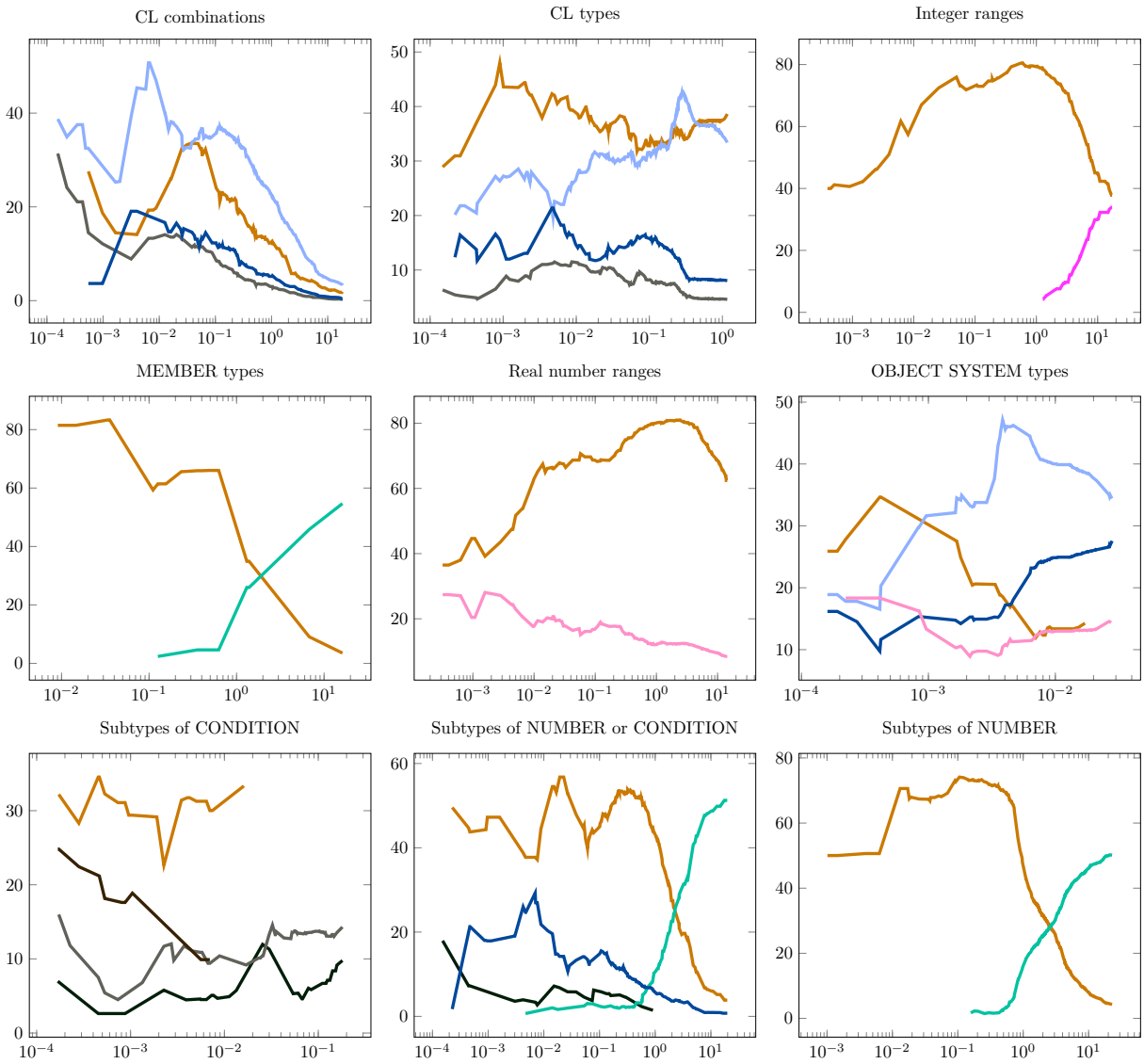
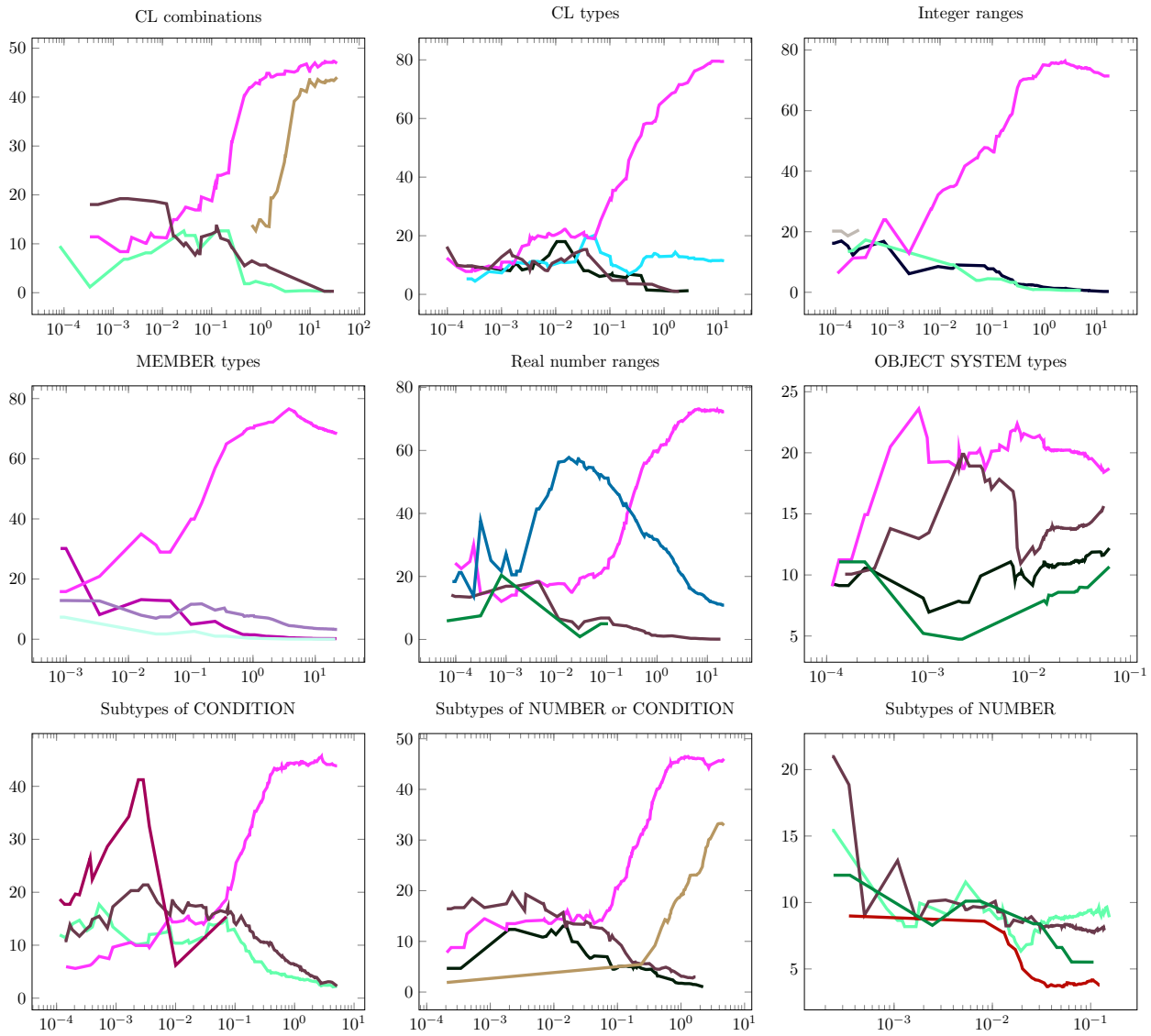
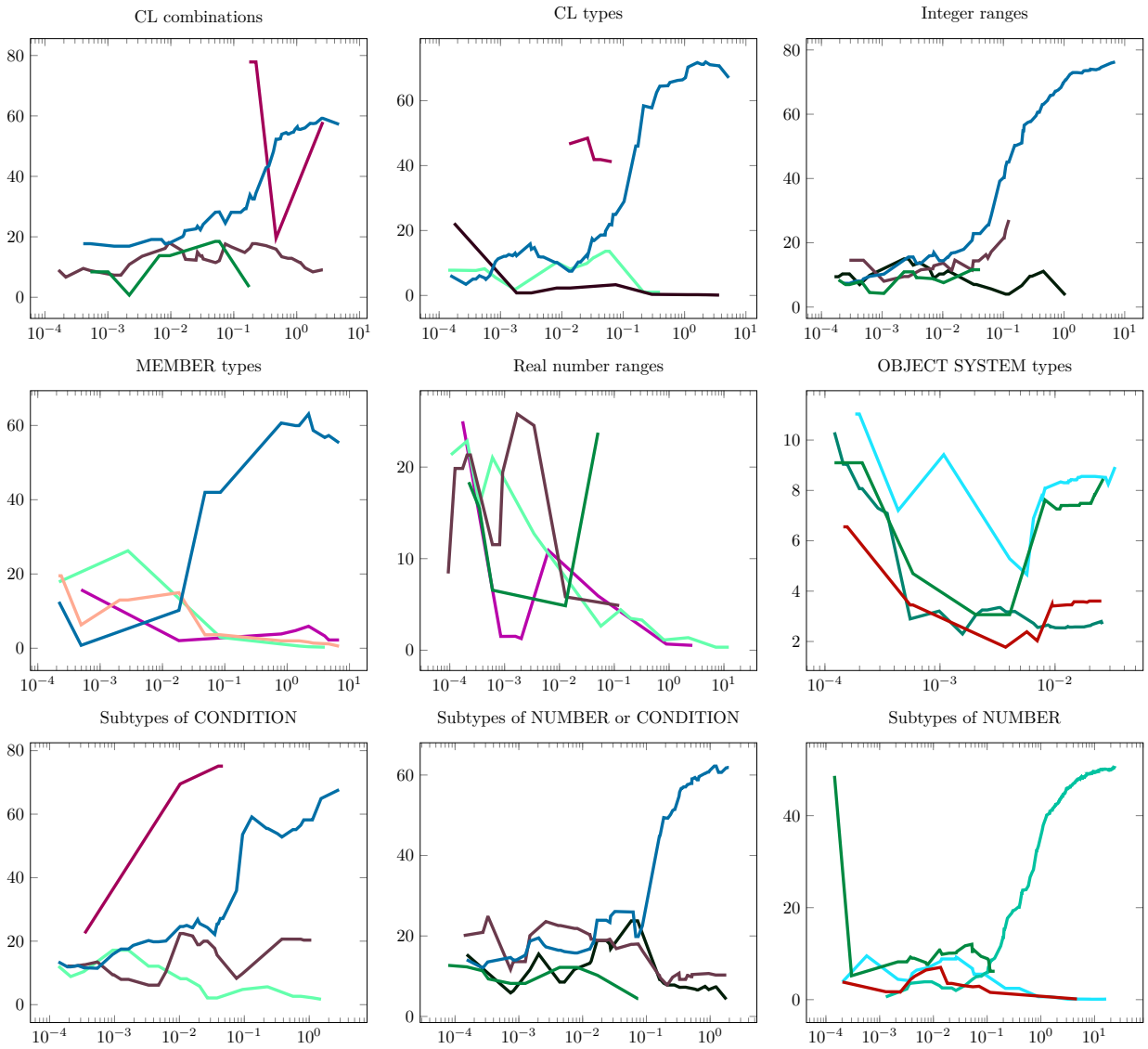


Figure 10.16: Performance Profile of various pools on algorithm **`mdtd-graph-baker`**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'



cached-subtypep-caching-call	4034.97 seconds	468,538,564 calls
smarter-subtypep-caching-call	384.37 seconds	19,379,929 calls
fixed-point	338.27 seconds	381,868,289 calls
subtypep-wrapper	162.75 seconds	35,428,011 calls
reduce-member-type	120.69 seconds	3,368,674 calls
bdd-find	96.04 seconds	50,847,858 calls
alphabetize-type	71.21 seconds	178,442,950 calls
slow-disjoint-types-p	52.04 seconds	3,081,716 calls
cached-subtypep	30.70 seconds	59,941,770 calls
alphabetize	27.20 seconds	16,469,600 calls
bdd-cmp	20.87 seconds	21,340,485 calls
type-to-dnf-bottom-up	20.71 seconds	50,127,911 calls
bdd-subtypep	7.68 seconds	1,441,792 calls
remove-supers	3.92 seconds	1,231,873 calls
bdd-dnf-wrap	3.53 seconds	297,928 calls

Figure 10.17: Performance Profile of various pools on algorithm **mdtd-bdd-strong**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'



delete-green-line	1134.48 seconds	3,960,437 calls
subtypep-wrapper	413.81 seconds	294,832,294 calls
fixed-point	177.43 seconds	227,498,120 calls
type-to-dnf-bottom-up	57.64 seconds	142,812,982 calls
bdd-find	48.70 seconds	25,059,561 calls
cached-subtypep-caching-call	36.12 seconds	44,121,832 calls
alphabetize-type	31.66 seconds	60,034,917 calls
slow-disjoint-types-p	28.16 seconds	704,048 calls
cached-subtypep	18.87 seconds	23,371,840 calls
bdd-find-int-int	15.04 seconds	6,246,851 calls
nodes	13.06 seconds	4,947,104 calls
alphabetize	9.96 seconds	7,857,928 calls
remove-supers	6.69 seconds	3,020,083 calls
delete-blue-arrow	2.98 seconds	92,314 calls

Figure 10.18: Performance Profile of various pools on algorithm **mdtd-bdd-graph**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'

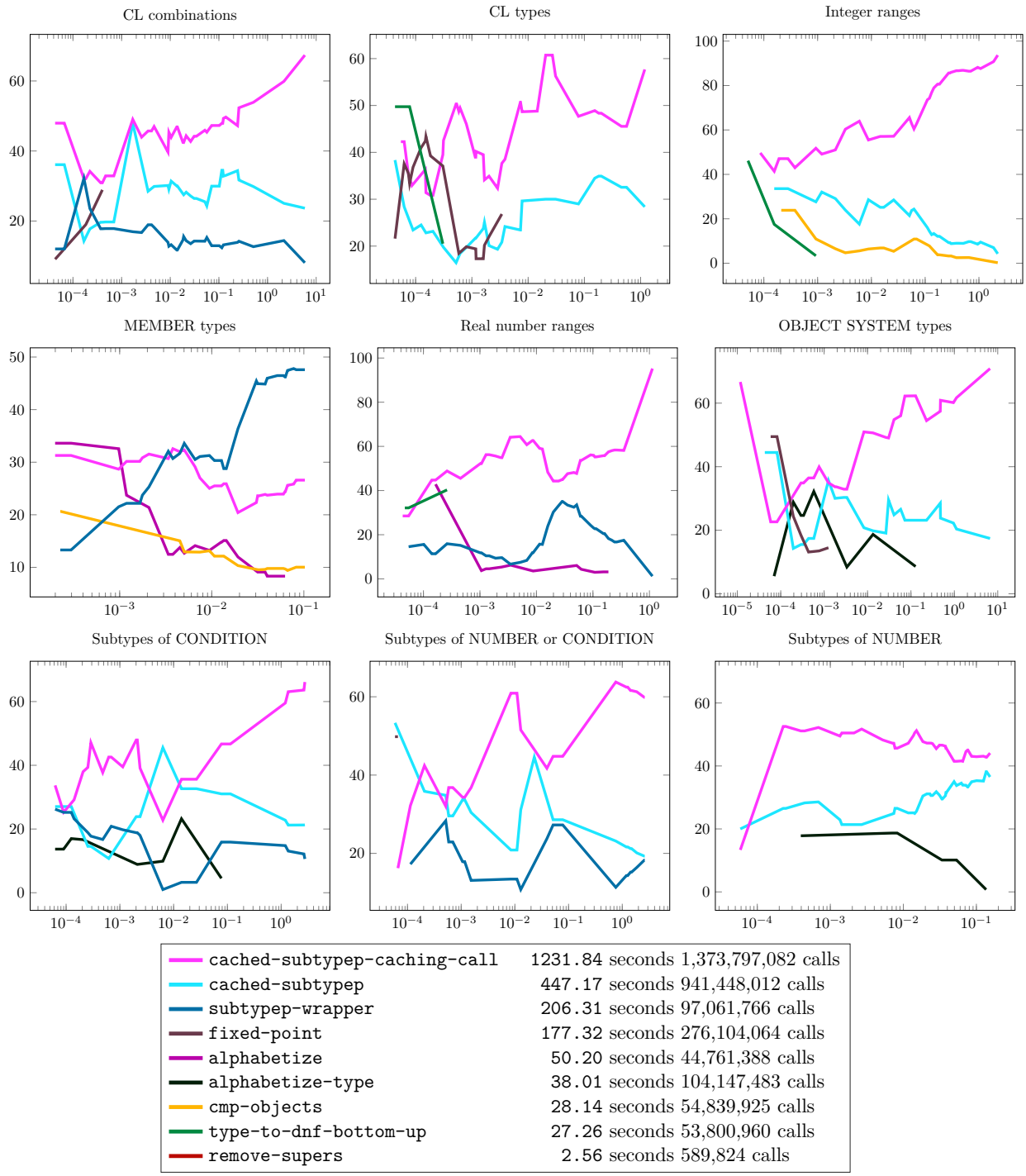
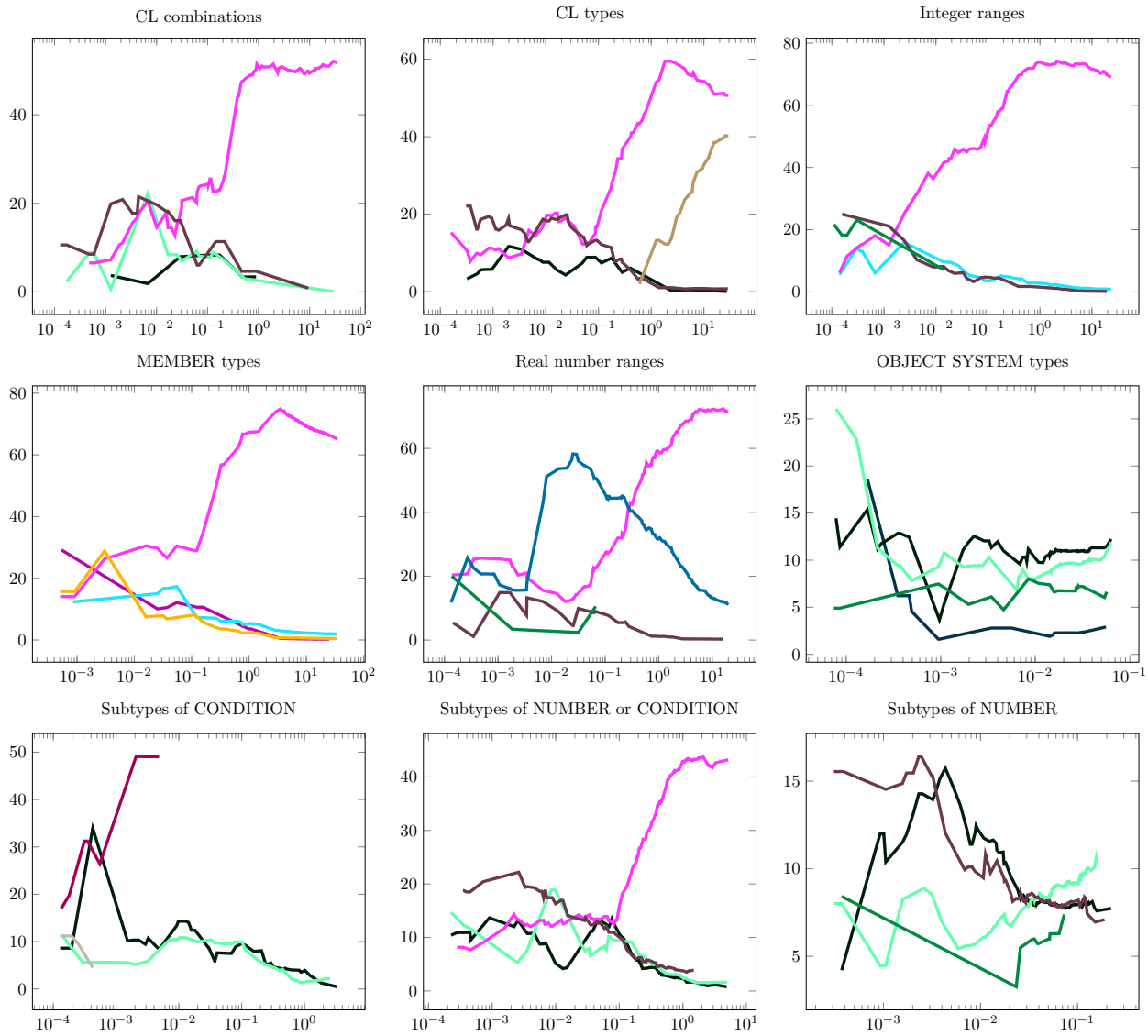


Figure 10.19: Performance Profile of various pools on algorithm **`mdtd-baseline`**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'



cached-subtypep-caching-call	3997.29 seconds	343,970,542 calls
fixed-point	245.71 seconds	293,296,284 calls
alphabetize-type	163.99 seconds	354,143,661 calls
subtypep-wrapper	140.58 seconds	28,167,004 calls
cached-subtypep	119.99 seconds	180,421,514 calls
smarter-subtypep-caching-call	118.87 seconds	5,893,142 calls
bdd-find	83.11 seconds	48,542,792 calls
slow-disjoint-types-p	39.74 seconds	1,796,096 calls
cmp-objects	34.02 seconds	183,133,460 calls
type-to-dnf-bottom-up	23.87 seconds	44,726,268 calls
alphabetize	15.27 seconds	16,499,045 calls
%bdd-to-dnf	9.56 seconds	1,655,040 calls
bdd-subtypep	5.89 seconds	2,162,688 calls

Figure 10.20: Performance Profile of various pools on algorithm **mdtd-bdd-weak**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'

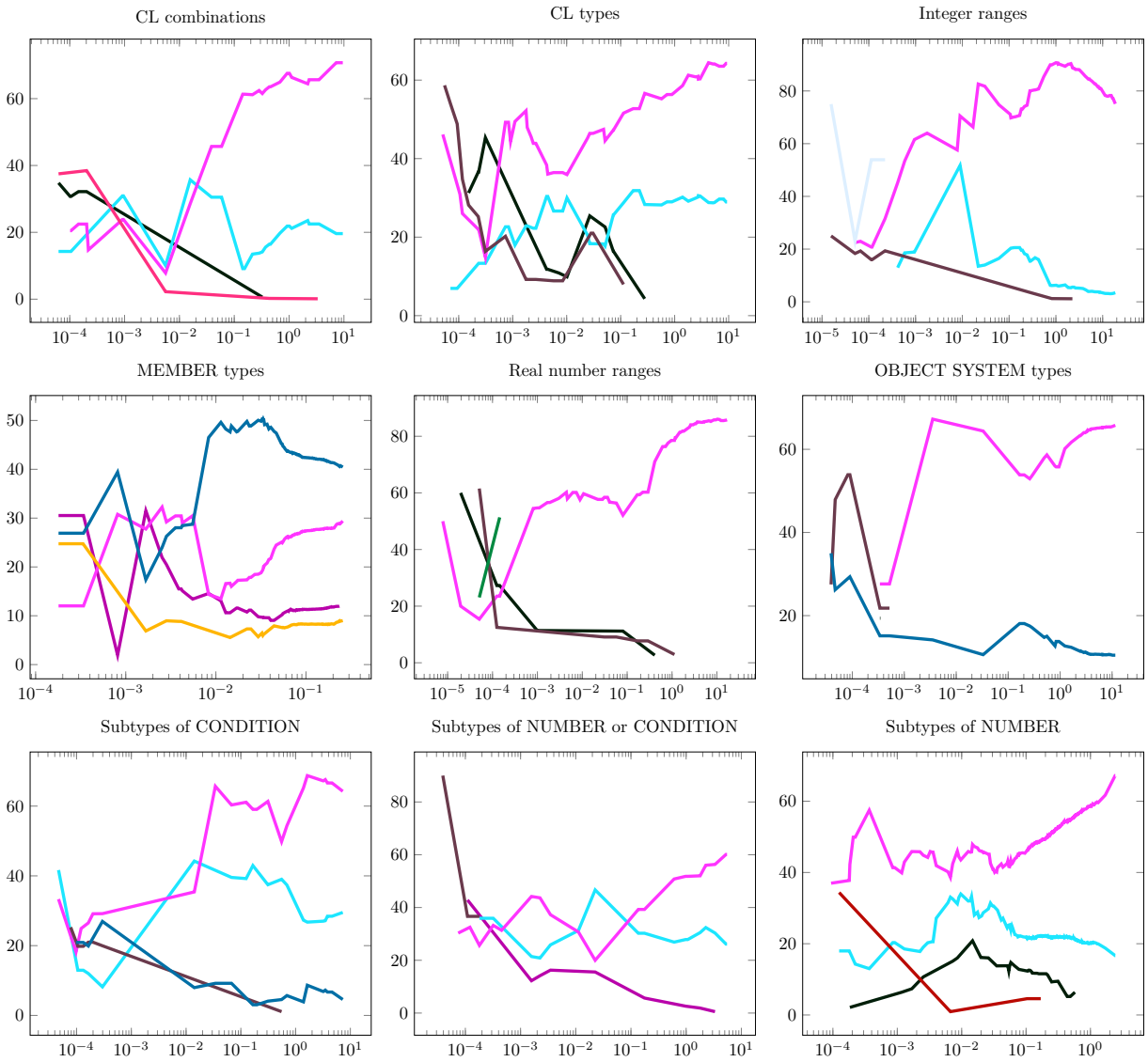


Figure 10.21: Performance Profile of various pools on algorithm **`mdtd-rtev2`**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'

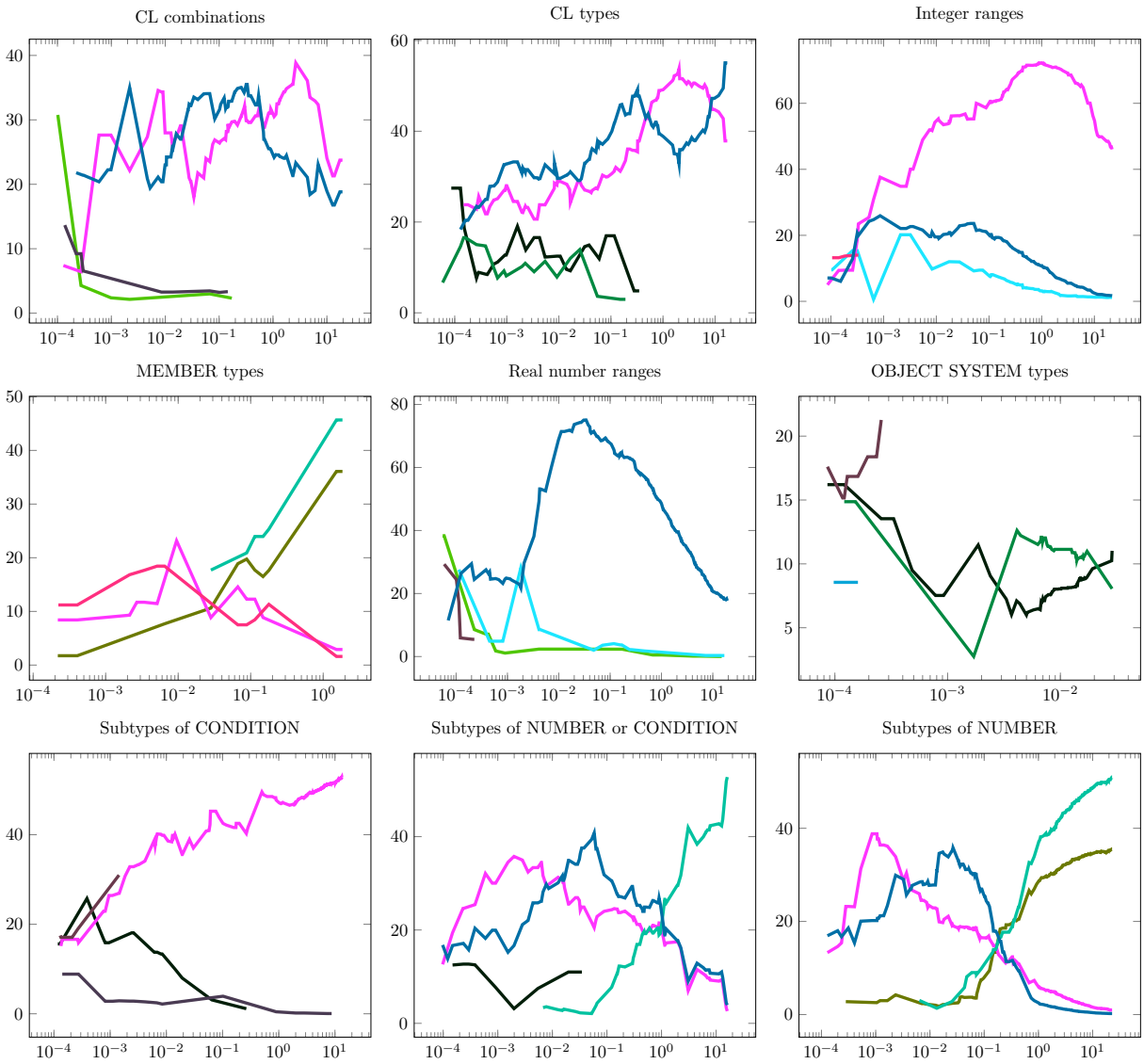


Figure 10.22: Performance Profile of various pools on algorithm **`mdtd-graph`**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'

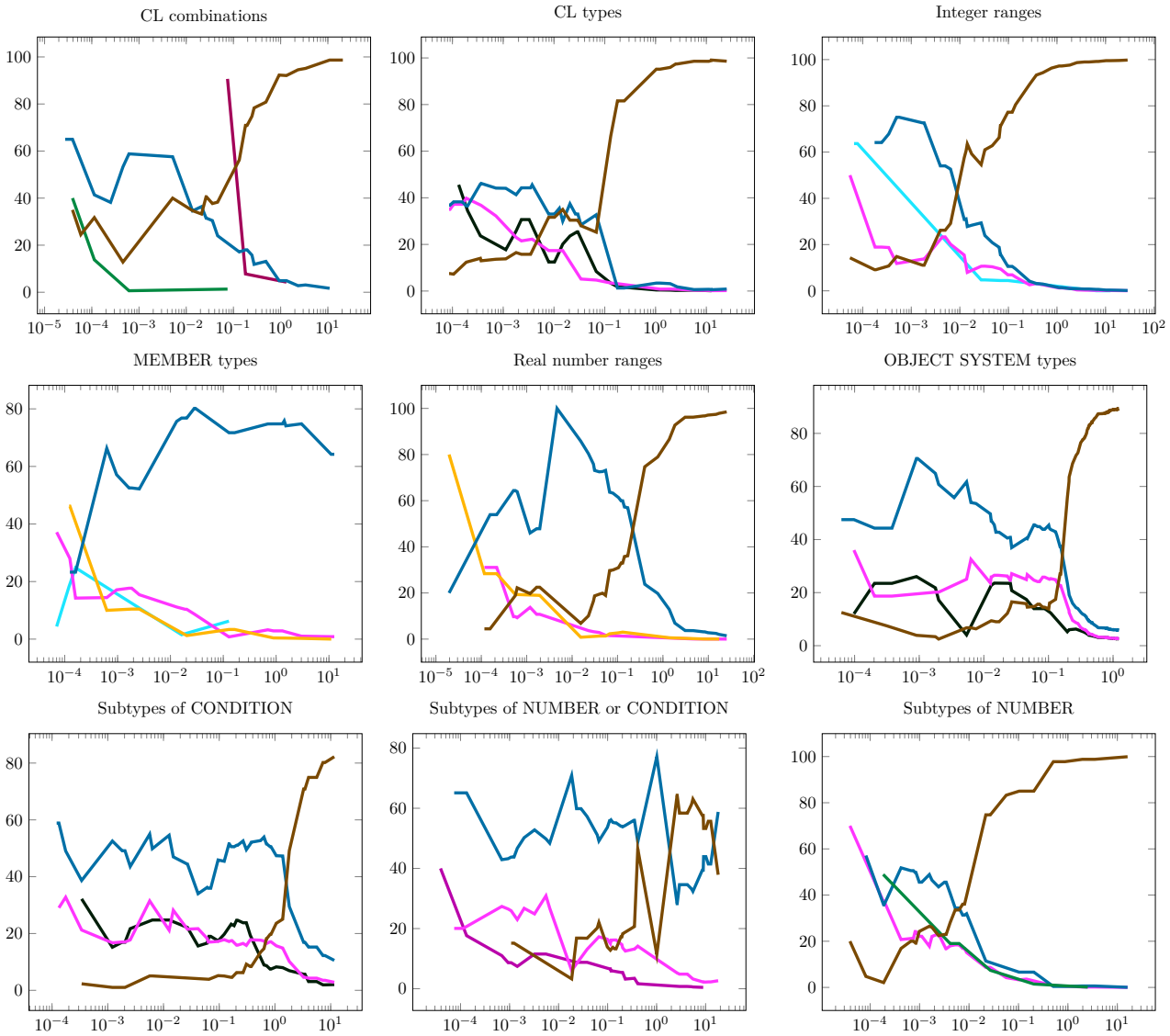
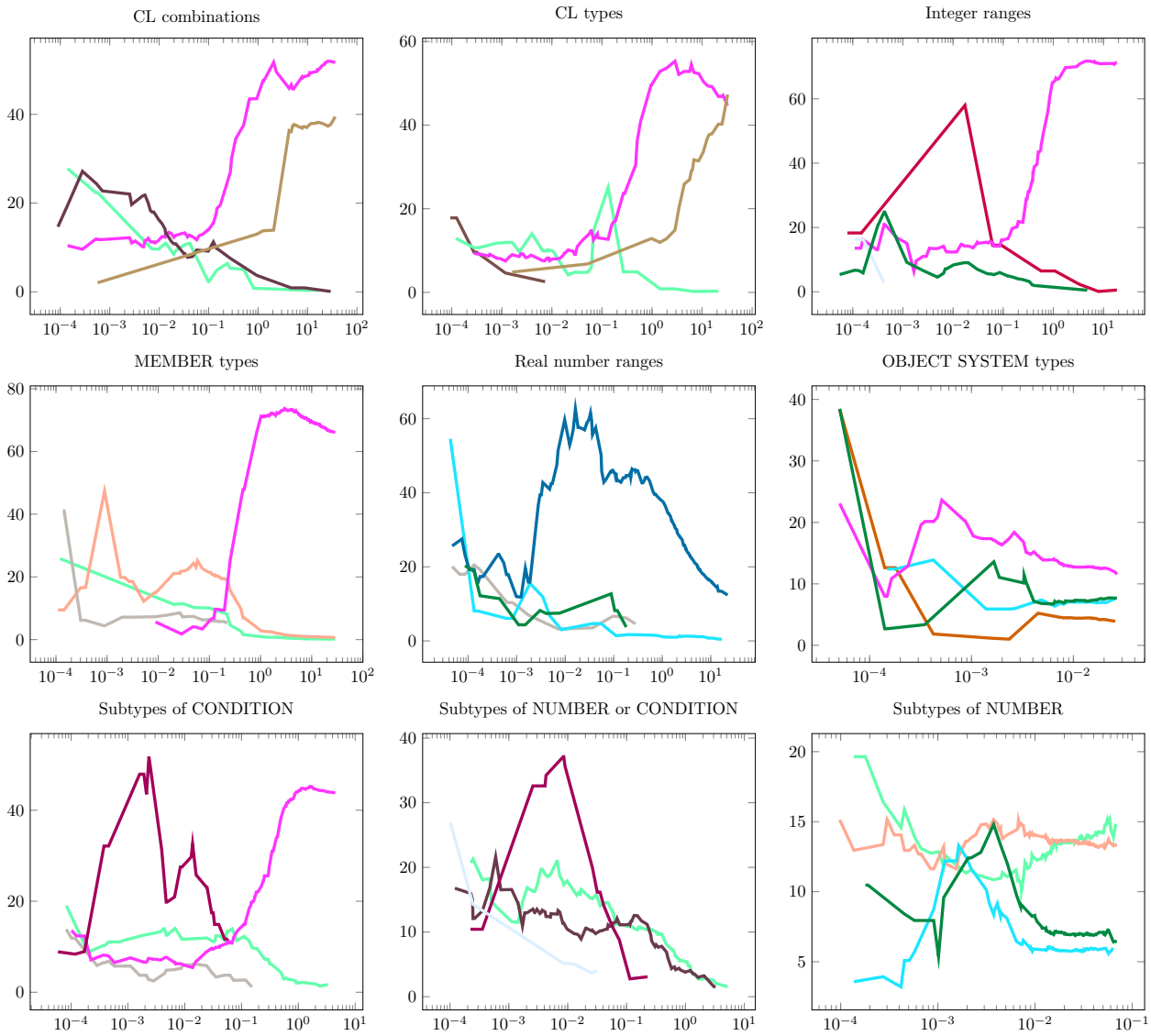


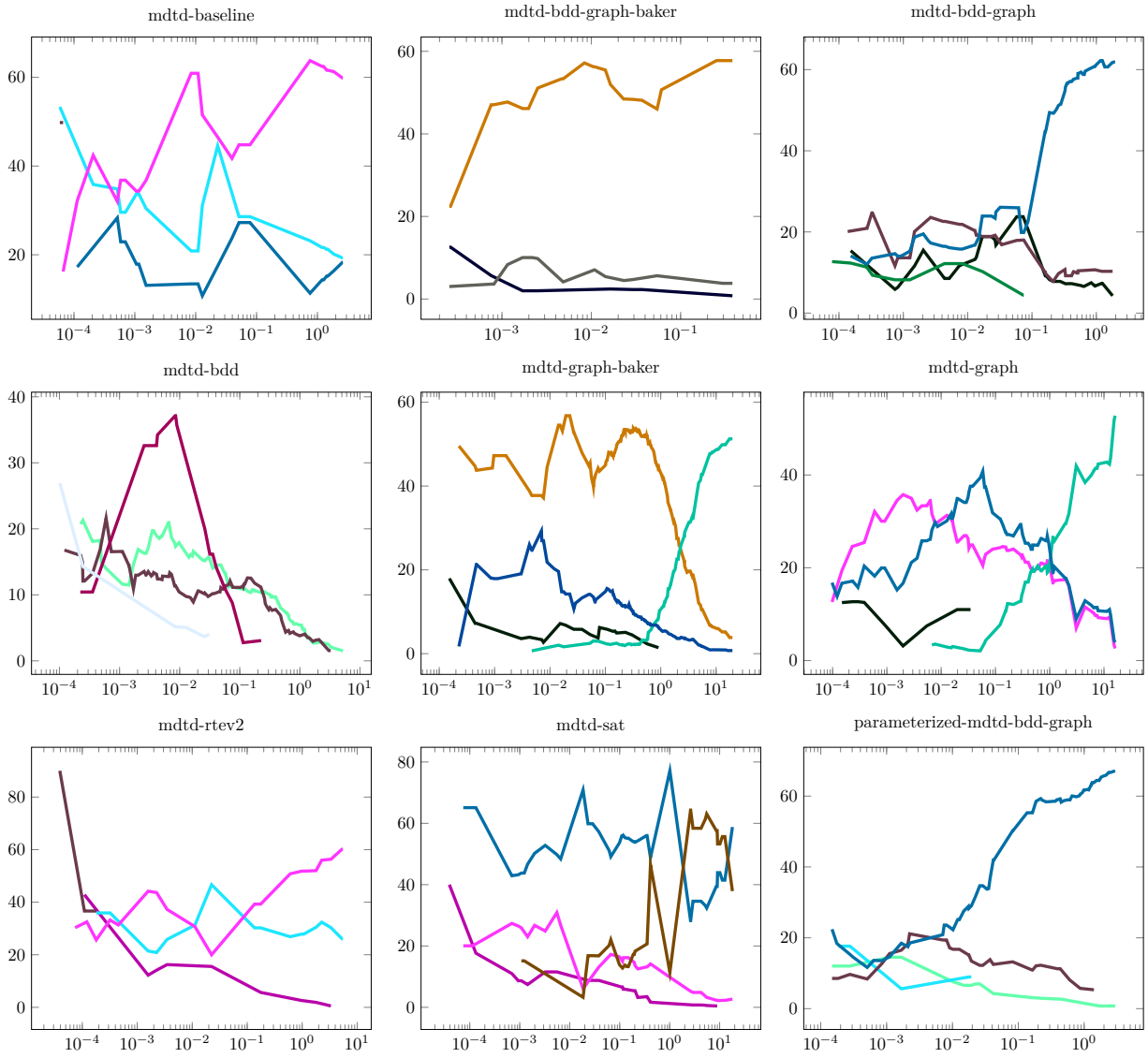
Figure 10.23: Performance Profile of various pools on algorithm **`mdtd-sat`**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'



cached-subtypep-caching-call	3620.18	seconds 384,898,542 calls
smarter-subtypep-caching-call	267.60	seconds 12,262,527 calls
bdd-find	259.47	seconds 157,151,513 calls
bdd-find-int-int	223.12	seconds 135,691,492 calls
subtypep-wrapper	202.45	seconds 56,555,993 calls
fixed-point	120.39	seconds 130,501,261 calls
type-to-dnf-bottom-up	106.80	seconds 226,012,171 calls
slow-disjoint-types-p	92.80	seconds 5,004,407 calls
cached-subtypep	81.41	seconds 91,352,909 calls
bdd-subtypep	57.25	seconds 52,760,741 calls
bdd-empty-type	19.03	seconds 13,217,744 calls
bdd-and-not	8.37	seconds 4,837,888 calls
type-to-dnf	8.16	seconds 4,645,376 calls
bdd-find-reduction	4.58	seconds 677,063 calls

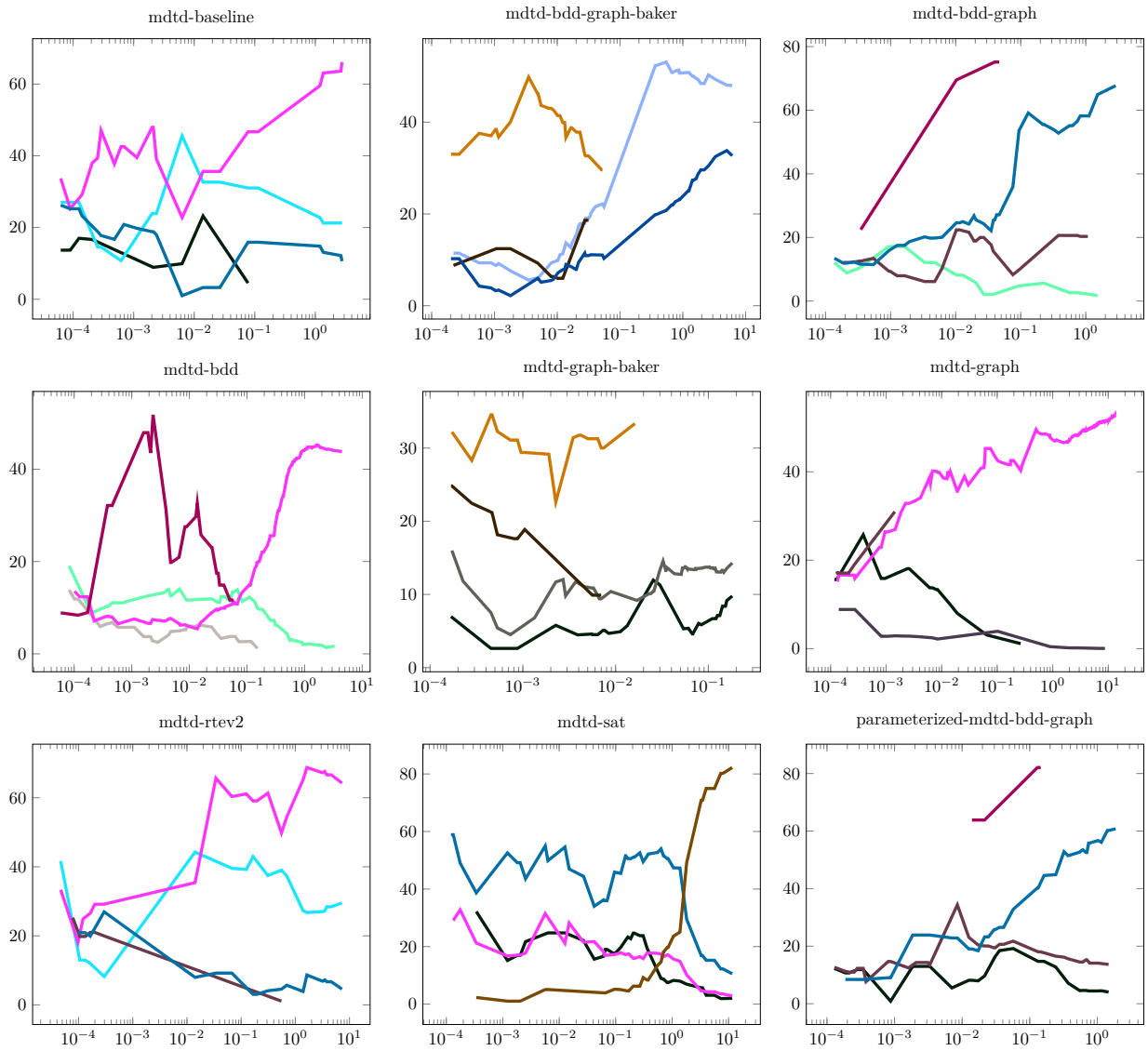
Figure 10.24: Performance Profile of various pools on algorithm **mdtd-bdd**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'

10.10 Profiler graphs of MDTD algorithms by function



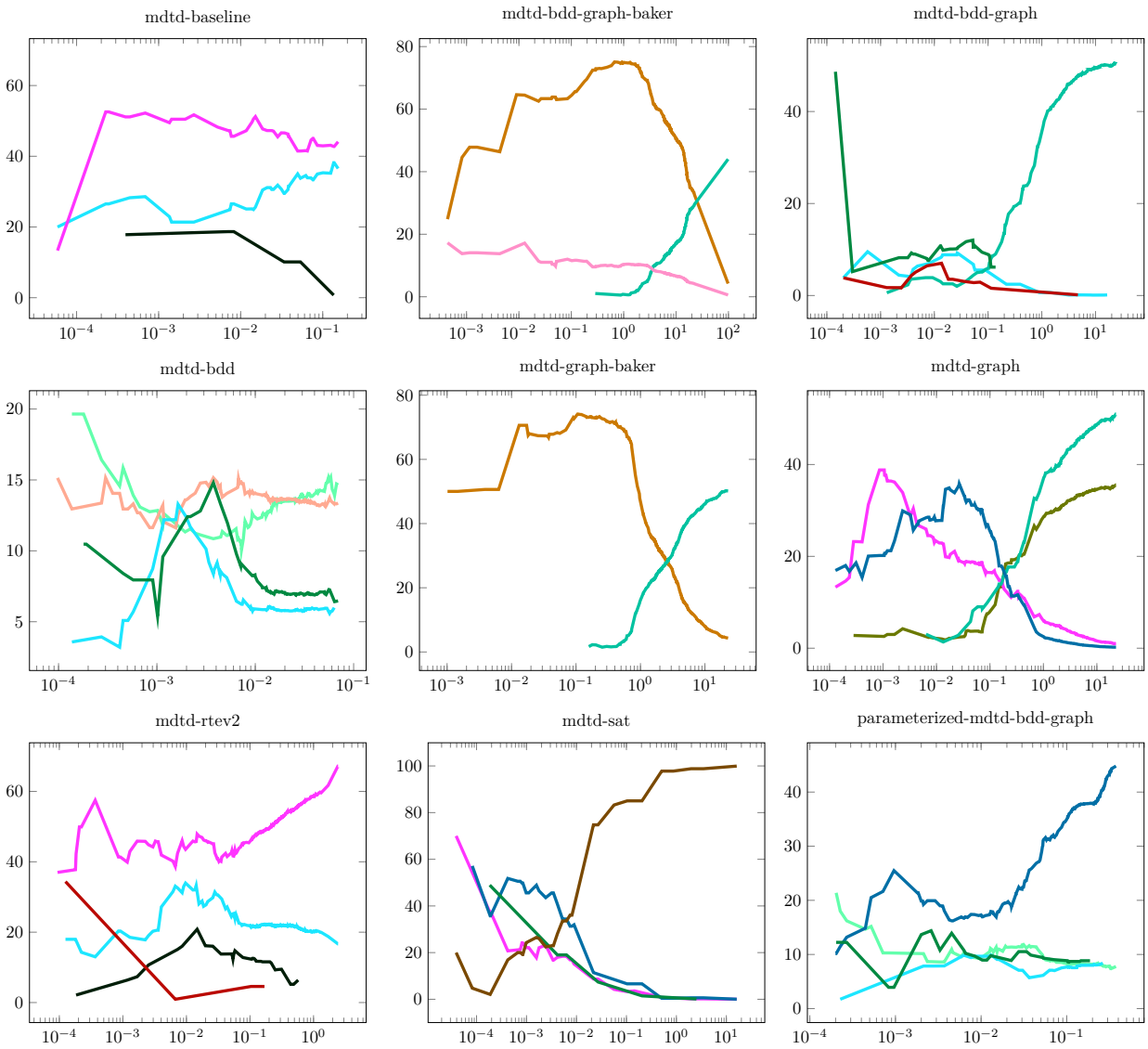
subtypep-wrapper	610.37 seconds	421,177,115 calls
cached-subtypep-caching-call	481.54 seconds	519,601,318 calls
fixed-point	389.31 seconds	475,652,506 calls
baker-subtypep	219.01 seconds	7,139,369 calls
delete-green-line	207.98 seconds	925,165 calls
cached-subtypep	119.01 seconds	223,358,784 calls
alphabetize-type	96.18 seconds	259,250,870 calls
visit-min-terms	76.96 seconds	32,970 calls
bdd-find	73.20 seconds	35,996,371 calls
type-keep-if	54.35 seconds	154,087,068 calls
slow-disjoint-types-p	38.66 seconds	3,450,639 calls
smarter-subtypep-caching-call	26.64 seconds	7,146,694 calls
alphabetize	21.82 seconds	39,594,264 calls
type-to-dnf-bottom-up	17.07 seconds	41,028,983 calls
type-to-dnf	12.76 seconds	3,959,296 calls
bdd-cmp	7.54 seconds	944,492 calls
bdd-empty-type	7.51 seconds	3,989,956 calls
recursively-expand-type	7.22 seconds	6,825,287 calls

Figure 10.25: Performance Profile of various MDTD functions for pool **Subtypes of NUMBER or CONDITION**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'



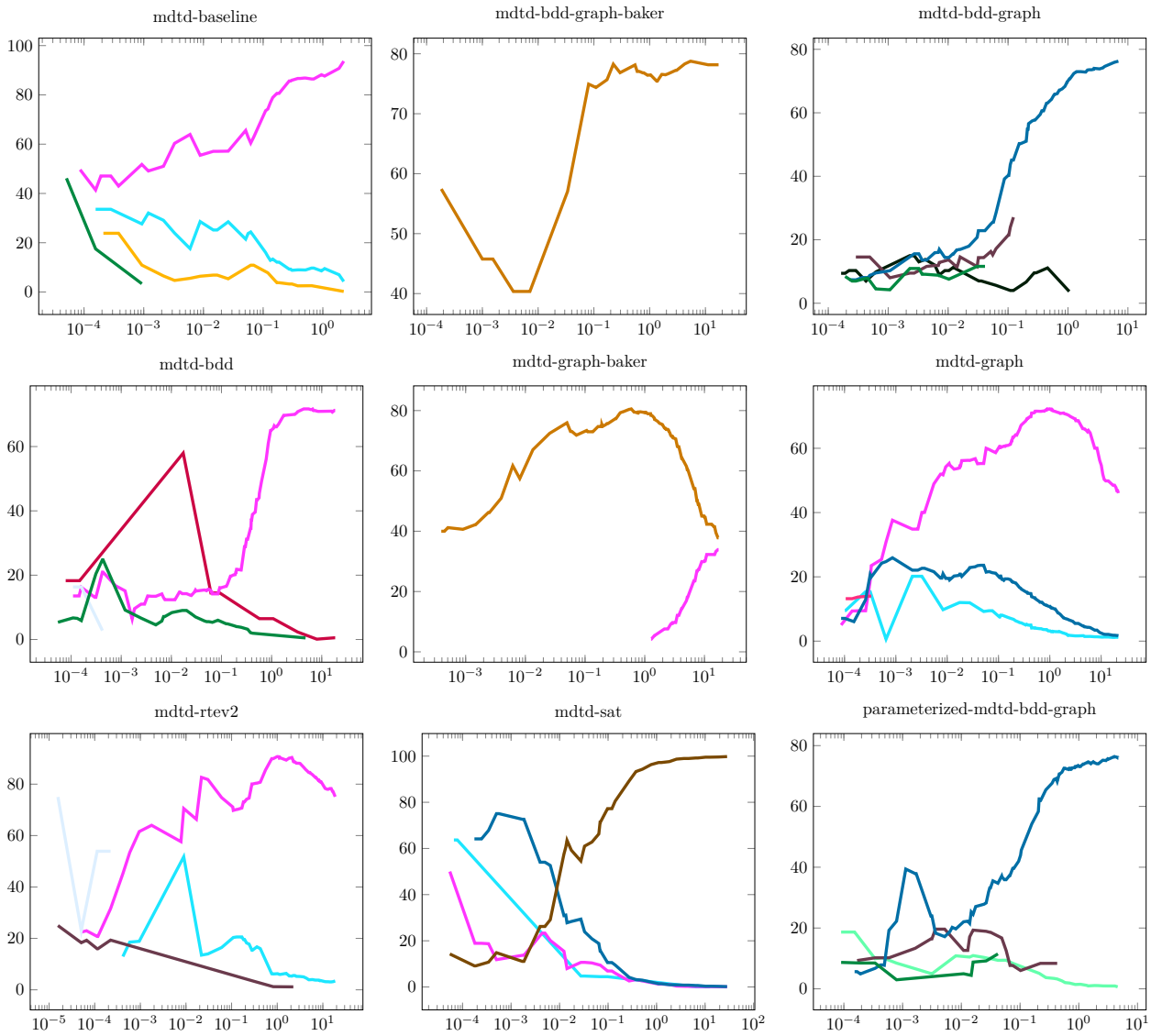
cached-subtypep-caching-call	1623.89	seconds 1,201,597,418 calls
fixed-point	341.47	seconds 335,897,010 calls
subtypep-wrapper	338.70	seconds 312,578,279 calls
slow-disjoint-types-p	248.14	seconds 13,522,698 calls
baker-subtypep	191.18	seconds 16,450,412 calls
alphabetize-type	143.57	seconds 356,862,306 calls
bdd-find	95.59	seconds 43,864,743 calls
bdd-subtypep	51.78	seconds 35,928,219 calls
visit-min-terms	49.40	seconds 99,620 calls
cached-subtypep	43.94	seconds 84,484,722 calls
literal-type-null?	41.68	seconds 6,393,731 calls
split-type	39.08	seconds 11,794,752 calls
recursively-expand-type	24.74	seconds 71,276,088 calls
type-keep-if	21.30	seconds 86,135,128 calls
extract-disjoint	11.96	seconds 600,326 calls
type-to-dnf	8.99	seconds 2,247,552 calls
type-to-dnf-bottom-up	6.12	seconds 5,649,280 calls
bdd	1.89	seconds 536,666 calls

Figure 10.26: Performance Profile of various MDTD functions for pool **Subtypes of CONDITION**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'



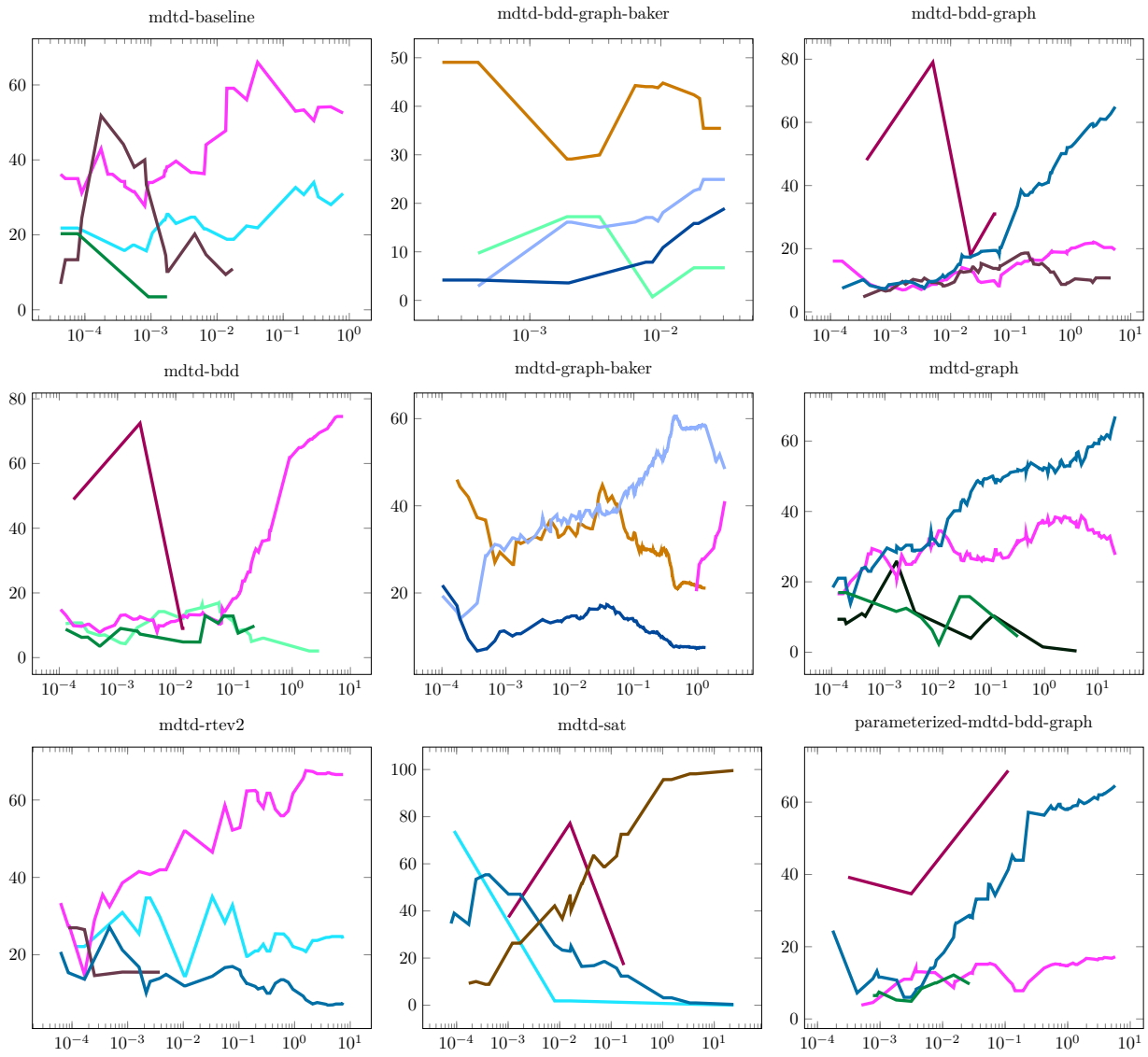
delete-green-line	7197.76 seconds	27,448,449 calls
baker-subtypep	2125.18 seconds	15,867,004 calls
add-green-line	1437.40 seconds	7,925,836 calls
cached-subtypep-caching-call	695.34 seconds	897,725,197 calls
subtypep-wrapper	607.62 seconds	429,904,149 calls
bdd-find-int-int	282.38 seconds	171,966,286 calls
bdd-find	243.65 seconds	156,208,662 calls
type/map-atomic-types	233.22 seconds	141,406,750 calls
cached-subtypep	229.71 seconds	504,729,363 calls
alphabetize-type	130.73 seconds	301,715,942 calls
fixed-point	114.04 seconds	176,857,050 calls
type-to-dnf-bottom-up	107.82 seconds	228,020,374 calls
visit-min-terms	81.96 seconds	515,171 calls
remove-supers	11.14 seconds	3,621,315 calls

Figure 10.27: Performance Profile of various MDTD functions for pool **Subtypes of NUMBER**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'



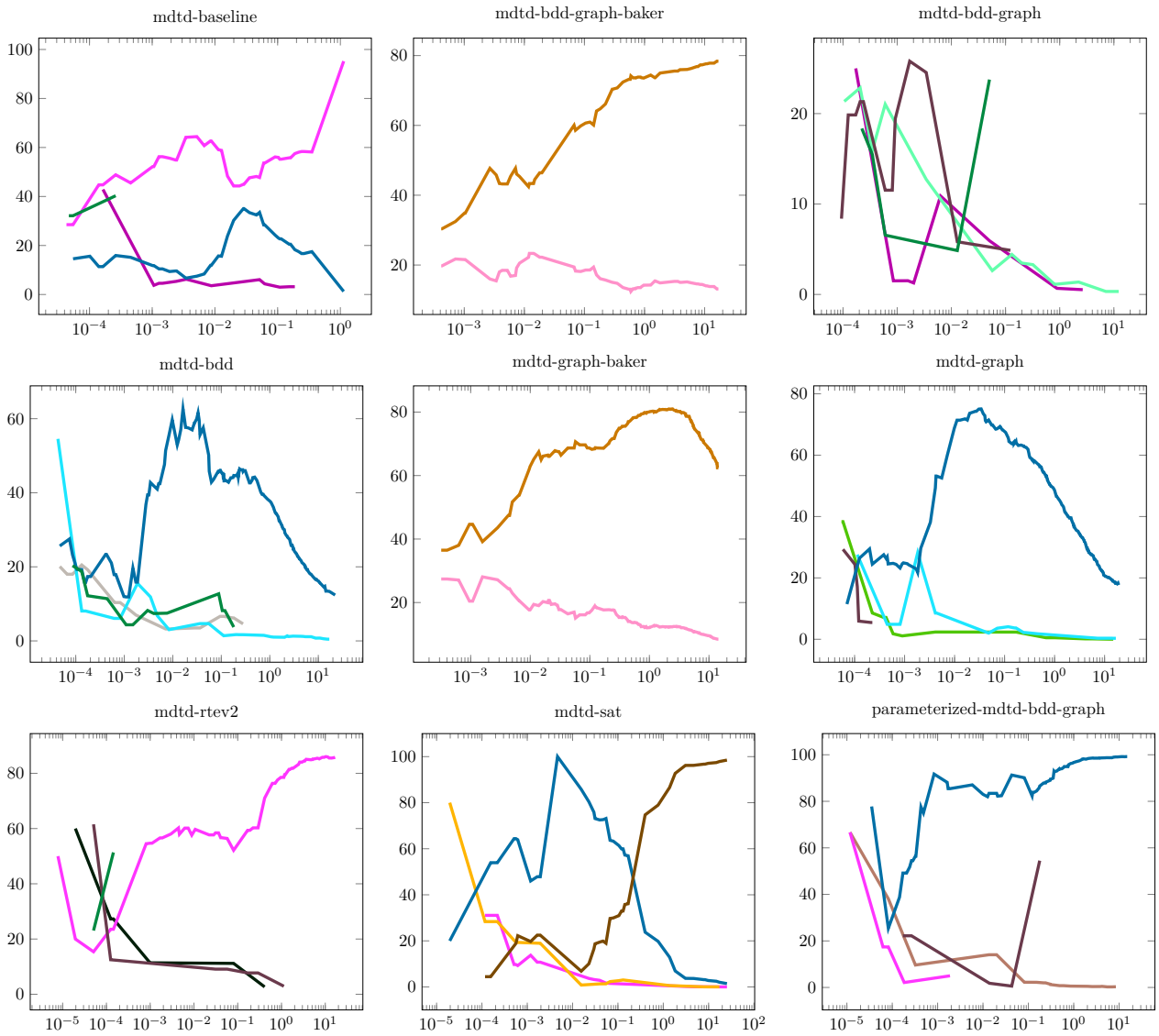
cached-subtypep-caching-call	2023.11 seconds	512,031,668 calls
subtypep-wrapper	899.19 seconds	580,984,545 calls
baker-subtypep	492.45 seconds	6,864,299 calls
fixed-point	182.48 seconds	177,900,614 calls
visit-min-terms	159.81 seconds	422,890 calls
cached-subtypep	114.34 seconds	200,924,808 calls
bdd-find	84.87 seconds	43,114,016 calls
type-to-dnf-bottom-up	69.41 seconds	122,030,540 calls
bdd-cmp	33.52 seconds	47,786,830 calls
alphabetize-type	29.44 seconds	64,562,959 calls
cmp-objects	17.21 seconds	17,248,589 calls
type-to-dnf	17.15 seconds	7,536,640 calls
bdd-subtypep	7.68 seconds	1,441,792 calls
reduce-lisp-type-once	7.17 seconds	2,752,512 calls
bdd-find-reduction	4.58 seconds	677,063 calls
add-green-line	3.84 seconds	127,087 calls

Figure 10.28: Performance Profile of various MDTD functions for pool **Integer ranges**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'



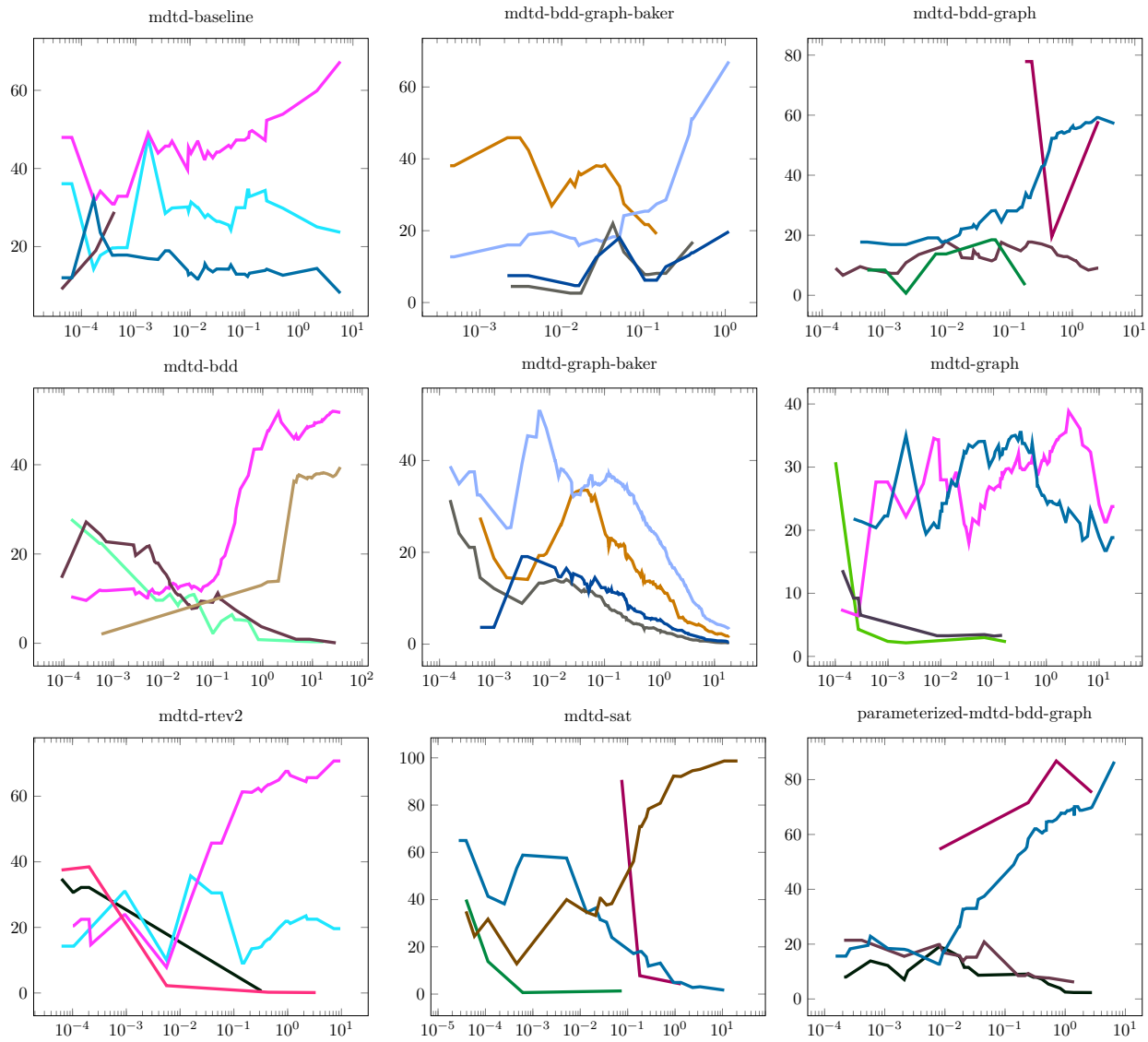
cached-subtypep-caching-call	1142.17 seconds	897,406,579 calls
subtypep-wrapper	798.96 seconds	519,603,537 calls
literal-type-null?	326.83 seconds	23,187,333 calls
fixed-point	254.03 seconds	338,599,029 calls
baker-subtypep	201.16 seconds	16,610,923 calls
cached-subtypep	117.42 seconds	226,930,280 calls
smarter-subtypep-caching-call	117.13 seconds	4,771,886 calls
slow-disjoint-types-p	97.36 seconds	4,001,355 calls
visit-min-terms	95.67 seconds	316,202 calls
bdd-find	65.87 seconds	45,630,488 calls
type-keep-if	57.30 seconds	215,133,640 calls
type-to-dnf-bottom-up	56.91 seconds	138,338,496 calls
alphabetize-type	46.87 seconds	141,924,785 calls

Figure 10.29: Performance Profile of various MDTD functions for pool **Subtypes of T**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'



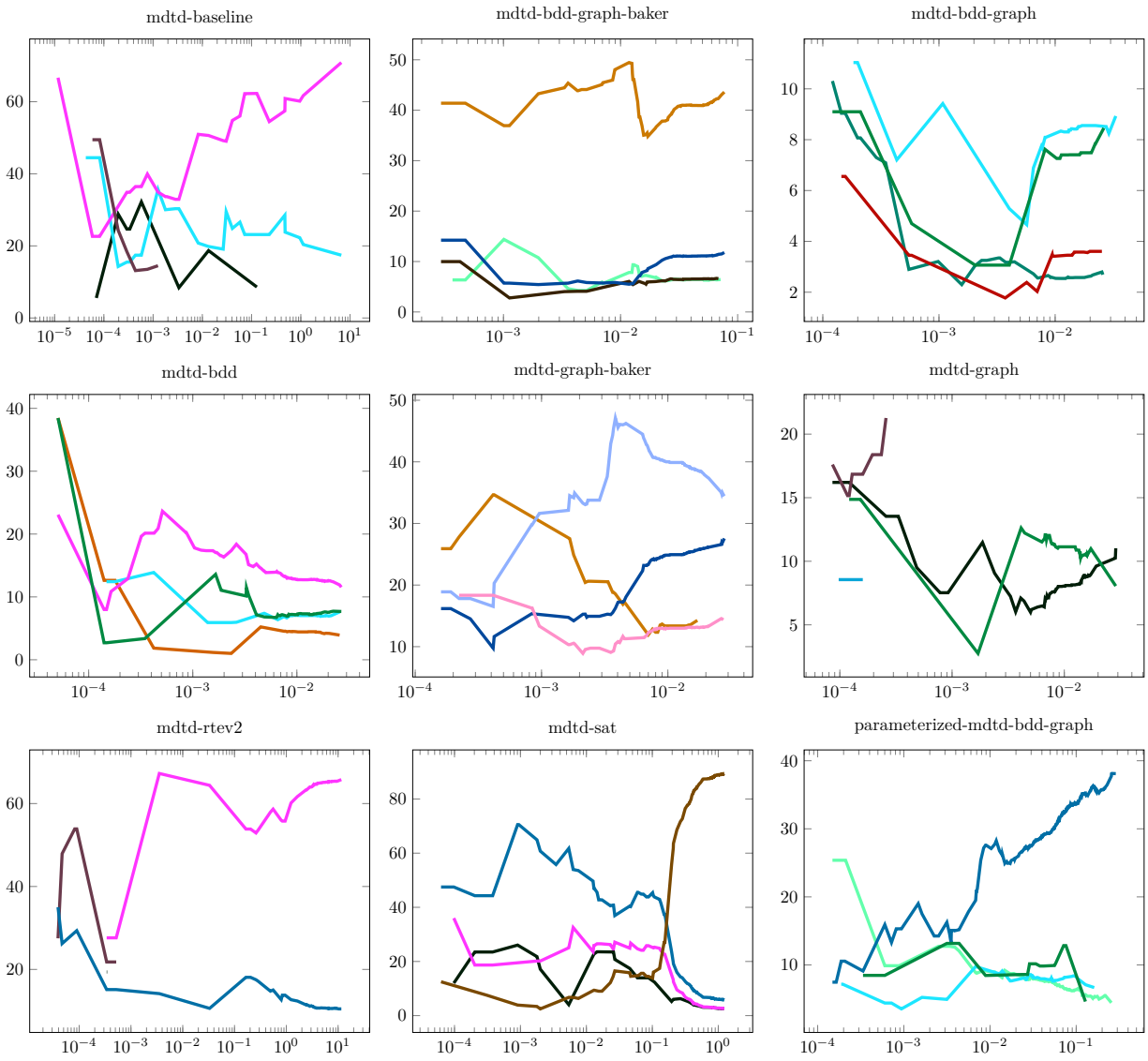
subtypep-wrapper	2762.24 seconds	1,343,320,914 calls
cached-subtypep-caching-call	1494.63 seconds	480,449,651 calls
baker-subtypep	826.46 seconds	9,184,127 calls
type/map-atomic-types	190.32 seconds	176,042,665 calls
visit-min-terms	126.97 seconds	364,835 calls
fixed-point	91.23 seconds	163,188,574 calls
type-to-dnf-bottom-up	50.64 seconds	82,310,760 calls
cmp-objects	33.20 seconds	51,746,888 calls
bdd-subtypep	27.86 seconds	29,255,416 calls
bdd-positive	24.17 seconds	29,775,898 calls
bdd-find	21.93 seconds	9,388,976 calls
cached-subtypep	20.50 seconds	31,908,677 calls
bdd-make-key	18.01 seconds	32,608,213 calls
alphabetize-type	12.27 seconds	42,197,202 calls
alphabetize	12.19 seconds	18,595,105 calls
(setf disjoint)	9.73 seconds	3,443,408 calls
remove-supers	4.62 seconds	236,156 calls
maybe-disjoint-node	3.24 seconds	419,396 calls

Figure 10.30: Performance Profile of various MDTD functions for pool **Real number ranges**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'



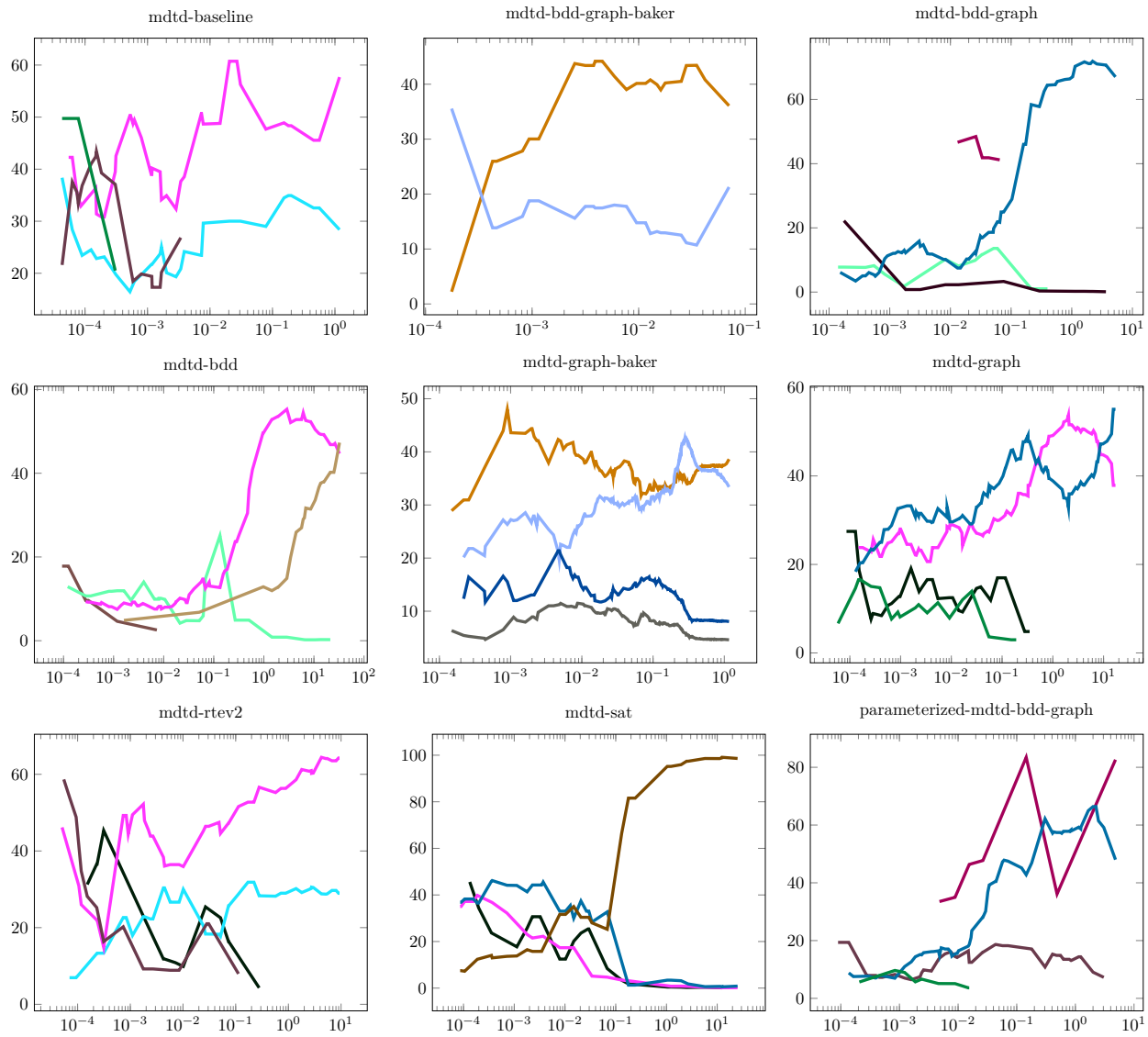
cached-subtypep-caching-call	1305.77 seconds	353,914,420 calls
smarter-subtypep-caching-call	576.26 seconds	22,892,758 calls
subtypep-wrapper	461.75 seconds	338,014,869 calls
fixed-point	171.70 seconds	266,302,751 calls
literal-type-null?	143.54 seconds	13,970,322 calls
cached-subtypep	88.29 seconds	198,958,982 calls
baker-subtypep	80.39 seconds	5,721,862 calls
visit-min-terms	52.92 seconds	263,473 calls
bdd-find	44.11 seconds	29,014,187 calls
alphabetize-type	41.17 seconds	117,837,159 calls
recursively-expand-type	31.73 seconds	41,961,448 calls
slow-disjoint-types-p	13.24 seconds	415,709 calls
type-to-dnf-bottom-up	12.36 seconds	62,358,410 calls
type-keep-if	12.15 seconds	71,292,276 calls
reduce-lisp-type-once	6.79 seconds	1,677,868 calls
extract-disjoint	6.51 seconds	328,006 calls
(setf disjoint)	3.47 seconds	462,213 calls

Figure 10.31: Performance Profile of various MDTD functions for pool **CL combinations**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'



cached-subtypep-caching-call	899.97 seconds	750,511,837 calls
subtypep-wrapper	580.46 seconds	443,445,701 calls
literal-type-null?	499.85 seconds	75,249,056 calls
fixed-point	444.81 seconds	547,301,581 calls
visit-min-terms	387.20 seconds	158,416 calls
alphabetize-type	312.71 seconds	933,747,342 calls
type-keep-if	300.20 seconds	888,806,768 calls
baker-subtypep	276.90 seconds	18,413,272 calls
bdd-find	186.38 seconds	87,524,776 calls
cached-subtypep	116.12 seconds	219,997,577 calls
type/map-atomic-types	109.64 seconds	235,825,120 calls
type-to-dnf-bottom-up	106.57 seconds	295,853,952 calls
bdd-empty-type	19.03 seconds	13,217,744 calls
split-type	17.14 seconds	7,267,216 calls
nodes	13.06 seconds	4,947,104 calls
%bdd-to-dnf	9.56 seconds	1,655,040 calls
bdd-and-not	6.66 seconds	2,818,048 calls
remove-supers	4.89 seconds	2,239,812 calls
call-with-subtype-hash	1.79 seconds	196,608 calls

Figure 10.32: Performance Profile of various MDTD functions for pool **OBJECT SYSTEM types**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'



cached-subtypep-caching-call	1263.14 seconds	837,922,902 calls
subtypep-wrapper	691.21 seconds	537,065,913 calls
fixed-point	319.78 seconds	483,037,881 calls
baker-subtypep	317.02 seconds	21,130,787 calls
smarter-subtypep-caching-call	282.67 seconds	13,928,226 calls
literal-type-null?	270.23 seconds	26,602,457 calls
alphabetize-type	182.76 seconds	483,007,703 calls
cached-subtypep	152.02 seconds	354,115,905 calls
visit-min-terms	108.69 seconds	551,892 calls
type-keep-if	71.38 seconds	260,639,652 calls
type-to-dnf-bottom-up	45.33 seconds	134,863,662 calls
bdd-find	41.86 seconds	27,342,516 calls
recursively-expand-type	37.25 seconds	85,716,308 calls
bdd-ident	17.68 seconds	25,351,606 calls
slow-disjoint-types-p	16.81 seconds	712,138 calls
bdd-and-not	8.37 seconds	4,837,888 calls
delete-blue-arrow	2.98 seconds	92,314 calls

Figure 10.33: Performance Profile of various MDTD functions for pool **CL types**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'

cached-subtypep-caching-call	10013.88 seconds	580,732,969 calls
subtypep-wrapper	1550.59 seconds	396,617,722 calls
cmp-objects	276.81 seconds	1,241,946,196 calls
alphabetize	150.20 seconds	129,864,599 calls
reduce-member-type	120.69 seconds	3,368,674 calls
cached-subtypep	86.45 seconds	117,539,645 calls
bdd-find-int-int	81.39 seconds	37,414,865 calls
bdd-find	45.51 seconds	26,441,795 calls
bdd-subtypep	35.77 seconds	37,976,713 calls
bdd-cmp	28.20 seconds	37,245,652 calls
baker-subtypep	18.33 seconds	32,865 calls
bdd-find-reduction	16.40 seconds	482,046 calls
delete-green-line	14.63 seconds	76,332 calls
maybe-disjoint-node	8.06 seconds	1,690,981 calls
reduce-lisp-type-once	7.99 seconds	7,030,799 calls
bdd-hash	7.89 seconds	18,915,531 calls
bdd-dnf-wrap	3.53 seconds	297,928 calls
subsets	2.82 seconds	327,680 calls
add-green-line	2.59 seconds	310,728 calls

Figure 10.34: Performance Profile of various MDTD functions for pool **MEMBER types**. Each plot is displayed with y='Profile Percentage' *vs.* x='Computation Time (seconds).'

10.11 Related work

Zabell [Zab08] explains that the unusual name, Student score, is used because William Sealy Gosset, who introduced the statistical techniques related to this quantity, published his results under the pseudonym, "Student." Gosset, who worked for Arthur Guinness, Son & Co., Ltd. in Dublin, Ireland (à la Guinness beer) in 1899, pioneered the use of statistical methods of small samples as part of the responsibilities of his job. Employees of Guinness were forbidden to publish, perhaps for fear of revealing trade secrets, but by enlightened contrast, they were allowed to take leave for study. Gosset convinced Guinness to allow him to publish certain results, as long as he didn't reveal information of any practical use to competitors. There is some dispute as to why Guinness demanded the pseudonym. Zabell claims that it is unknown whether Guinness did not want their competitors to know that they hired statisticians, or whether the pseudonym was to protect Gosset's identity from other Guinness employees. Zabell cites Hotelling [Hot30] for these claims.

BDDs have been used in electronic circuit generation [CBM90], verification, symbolic model checking [BCM+92], and type system models such as in XDuce [HVP05]. None of these sources discuss how to extend the BDD representation to support subtypes.

Decision tree techniques are useful in the efficient compilation of pattern-matching constructs in functional languages [Mar08]. An important concern in pattern-matching compilation is that finding the best ordering of the variables is known to be coNP-Complete [Bry86]. However, when using BDDs to represent Common Lisp type specifiers, we obtain representation (pointer) equality simply by using a consistent ordering; finding the *best* ordering is not necessary for our application.

The average and worst cases of decision diagrams have been discussed in published works. For example Butler *et al.* [SIHB97] discuss these measurements for symmetric multiple-valued functions. Bryant [Bry86] discusses the effect of variable ordering on diagram sizes. Gröpl [GPS01] examines the worst case size of qOBDDs, which are quasi-reduced BDDs; *i.e.* he omits some of the reduction steps which we use.

Castagna [Cas16] discusses the use of BDDs as a tool for type manipulation primarily in strictly typed (or gradually typed [CL17]) functional languages. A principal motivation for our research was (is) to investigate how these techniques might be useful when applied to a more pragmatic type system such as in Common Lisp.

The Common Lisp specification [Ans94] defines the term *exhaustive partition* which is a concept related to the maximal disjoint decomposition. In terms of the definitions from Common Lisp, we may think of a disjoint decomposition of $V \subset \mathbb{P}(U)$ as an exhaustive partition of $\bigcap V$. The distinction between the two concepts is subtle. To find an exhaustive partition, we start with a single set and partition it into disjoint subsets whose union is the original set. To find a disjoint decomposition, we start with a set of possibly overlapping subsets of a given set, whose union is not necessarily the entire set, and we proceed by finding another set of subsets which is pairwise disjoint and which has the same union as the given set of subsets.

The problem of finding the maximal disjoint decomposition is similar to the *union find* problem [PBM10, GF64]. In union find, we are permitted to look into the sets and partition the elements. However, in MDTD we wish to form the partition without knowledge of the specific elements; *i.e.* we are not permitted to iterate over or visit the individual elements. Rather, we have knowledge of the subset, superset, disjointness relations between any pair of sets.

The correspondence of types to sets and subtypes to subsets is a subject of ongoing research in computer science. A brief synopsis of the history is given in Section 2.8.

There seems to be a connection between MDTD and ISOP (Irredundant Sum-Of-Products generation) described by Minato [iM96]. More research is needed to categorize this connection.

10.12 Conclusion and perspectives

The results of the performance testing in this chapter lead us to believe that the BDD as data structure for representing Common Lisp type specifiers is promising. However, there is still work to do, especially in identifying heuristics to predict its performance relative to more traditional approaches.

It is known that algorithms using BDD data structure tend to trade space for speed; *i.e.*, allocating more space to get more speed. Castagna [Cas16] suggests a lazy version of the BDD data structure which may reduce the memory footprint which, in turn, would have a positive effect on the BDD based algorithms. We have spent only a few weeks optimizing our BDD implementation based on the Andersen's description [And99], whereas the CUDD [Som] developers have spent many years optimizing their algorithms. Certainly our BDD algorithm can be made more efficient using techniques of CUDD or others.

A shortcoming of our research is the lack of formal proof for the correctness of our algorithms, most notably the graph decomposition algorithm from Section 9.4.

It has also been observed that, in the algorithm explained in section 9.4, the convergence rate varies depending on the order of the reduction operations. We do not have sufficient data yet to characterize this dependence. Furthermore, the order to break connections in the algorithm in Section 9.4 may be important. It is clear that

many different strategies are possible, *e.g.*, (1) break busiest connections first, (2) break connections with the fewest dependencies, (3) random order, (4) closest to top of tree, etc. These are all areas of ongoing research.

We plan to investigate whether there are other applications of MDTD outside the Common Lisp type system. We hope that some users of Castagna's techniques [Cas16] of semantic subtyping will benefit from the optimizations we have discussed.

A potential application with Common Lisp is improving the `subtypep` implementation itself, which is known to be slow in some cases. Implementation 9.6 gave a BDD specific implementation of `bdd-subtypep`. We intend to investigate whether existing Common Lisp implementations could use our technique to represent type specifiers in their inferencing engines, and thereby make some subtype checks more efficient.

Chapter 11

Strategies for typecase Optimization

In Chapter 4 we introduced the problem of efficiently recognizing a sequence of objects in Common Lisp, given a regular type expression. This problem lead to two challenges called the MDTD problem and the serialization problem. In Chapter 9 we looked at solutions to the MDTD problem, and in Chapter 10 we analyzed the performance characteristics of various strategies. In the current chapter we will look at the serialization problem.

11.1 Introduction

The `typecase` macro is specified in Common Lisp [Ans94] as a run-time mechanism for selectively branching as a function of the type of a given expression. Example 11.1 summarizes the usage. The type specifiers used may be simple type names such as `fixnum`, `string`, or `my-class`, but may also specify more expressive types such as range checks `(float -3.0 3.5)`, membership checks such as `(member 1 3 5)`, arbitrary Boolean predicate checks such as `(satisfies oddp)`, or logical combinations of other valid type specifiers such as `(or string (and fixnum (not (eql 0))) (cons bignum))`. A more detailed discussion of the semantics of type specifiers can be found in Section 2.2.

Example 11.1 (Synopsis of `typecase` syntax)**.**

```
(typecase keyform
  (Type.1 body-forms-1...)
  (Type.2 body-forms-2...)
  (Type.3 body-forms-3...)
  ...
  (Type.n body-forms-n...))
```

In this chapter, we consider several issues concerning the compilation of such a `typecase` usage.

- Redundant checks[1] — The set of type specifiers used in a particular invocation of `typecase` may have subtype or intersection relations among them. Consequently, it is possible (perhaps likely in the case of auto-generated code) that the same type checks be performed multiple times, when evaluating the `typecase` at run-time.
- Unreachable code — The specification suggests but does not require that the compiler issue a warning if a clause is not reachable, being completely shadowed by earlier clauses. We consider such compiler warnings desirable, especially in manually written code.
- Exhaustiveness — The user is allowed to specify a set of clauses which is non-exhaustive. If it can be determined at compile time that the clauses are indeed exhaustive, even in the absence of a `t`/`otherwise` clause, then in such a case, the final type check may be safely replaced with `otherwise`, thus eliminating the need for that final type check at run-time.

The `etypecase` macro (exhaustive type case according to [Ste90, Section 29.4.3]) promises to signal a run-time error if the object is not an element of any of the specified types. We pose a different exhaustiveness

[1] Don't confuse redundant check with redundancy check. In this report we address the former, not the latter. A type check is viewed as redundant, and can be eliminated, if its Boolean result can determined by static code analysis.

question: can it be determined at compile time that all possible values are covered by at least one of the clauses?

Assuming we are allowed to change the `typecase` evaluation order, we wish to exploit evaluation orders which are more likely to result in faster run-time. Some type checks are slower than others. E.g. a `satisfies` check may be arbitrarily slow. Under certain conditions, as will be seen, there are techniques to protect certain type checks to allow reordering without changing semantics. Such reordering may consequently enable particular optimizations such as elimination of redundant checks or the exhaustiveness optimization explained above. Elimination of redundant type checks has an additional advantage that, apart from potentially speeding up certain code paths, it also allows the discovery of unreachable code.

There is a statement in the `typecase` specification that each *normal-clause* be considered in turn. We interpret this requirement not to mean that the type checks must be evaluated in order, but rather that each type test must assume that type tests appearing earlier in the `typecase` are not satisfied. Moreover, we interpret this specified requirement so as not to impose a run-time evaluation order and that, as long as evaluation semantics are preserved, the type checks may be done in any order at run-time, and in particular, that any type check which is redundant or unnecessary need not be performed.

In this chapter we consider different techniques for evaluating the type checks in different orders than that which is specified in the code, so as to maintain the semantics but to eliminate redundant checks.

In the chapter we examine two very different approaches for performing certain optimizations of `typecase`. First, we use a natural approach using s-expression based type specifiers (Section 11.2), operating on them as symbolic expressions. In the second approach (Section 11.3) we employ Reduced Ordered Binary Decision Diagrams (ROBDDs). We finish the chapter with an overview of related work (Section 11.4) and a summary of perspectives (Section 11.5).

11.2 Type specifier approach

We would like to automatically remove redundant checks such as `(eql 42)`, `(member 40 41 42)`, and `fixnum` in Example 11.2.

Example 11.2 (typecase with redundant type checks)**.**

```
(typecase OBJECT
  ((eql 42)
   body-forms-1...)
  ((and (member 40 41 42) (not (eql 42)))
   body-forms-2...)
  ((and fixnum (not (member 40 41 42)))
   body-forms-3...)
  ((and number (not fixnum))
   body-forms-4...))
```

The code in Example 11.3 is semantically identical to that in Example 11.2, because a type check is only reached if all preceding type checks have failed.

Example 11.3 (typecase after removing redundant checks)**.**

```
(typecase OBJECT
  ((eql 42) body-forms-1...)
  ((member 40 41 42) body-forms-2...)
  (fixnum body-forms-3...)
  (number body-forms-4...))
```

In the following sections, we initially show that certain duplicate checks may be removed through a technique called forward-substitution and reduction (Section 11.2.1). A weakness of this technique is that it sometimes fails to remove particular redundant type checks. Because of this weakness, a more elaborate technique is applied, in which we augment the type tests to make them mutually disjoint (Section 11.2.3). With these more complex type specifiers in place, the `typecase` has the property that its clauses are reorderable, which allows

the forward-substitution and reduction algorithm to search for an ordering permitting more thorough reduction (Section 11.2.5). This process allows us to identify unreachable code paths and to identify exhaustive case analyses, but there are still situations in which redundant checks cannot be eliminated.

11.2.1 Reduction of type specifiers

There are legitimate cases in which the programmer has specifically ordered the clauses to optimize performance. A production worthy `typecase` optimization system should take that into account. However, for the sake of simplicity, the remainder of this chapter ignores this concern.

We introduce a macro, `reduced-typecase`, which expands to a call to `typecase` but with cases reduced where possible. Latter cases assume previous type checks fail. This transformation preserves clause order, but may simplify the executable logic of some clauses. In the expansion, in Example 11.4 the second `float` check is eliminated, and consequently, the associated `AND` and `NOT`.

Example 11.4 (Simple invocation and expansion of `reduced-typecase`).

```
(reduced-typecase object
   (float body-forms-1...)
   ((and number (not float)) body-forms-2...))

(typecase object
   (float body-forms-1...)
   (number body-forms-2...))
```

To illustrate how this reduction works, we provide a sightly more elaborate example. In Example 11.5, the first type check is `(not (and number (not float)))`. In order that the second clause be reached at run-time the first type check must have already failed. This means that the second type check, `(or float string (not number))`, may assume that `object` is not of type `(not (and number (not float)))`.

Example 11.5 (Invocation and expansion `reduced-typecase` with unreachable code path).

```
(reduced-typecase object
   ((not (and number (not float))) body-forms-1...)
   ((or float string (not number)) body-forms-2...)
   (string body-forms-3...))

(typecase object
   ((not (and number (not float))) body-forms-1...)
   (string body-forms-2...)
   (nil body-forms-3...))
```

The `reduced-typecase` macro rewrites the second type test `(or float string (not number))` by a technique called forward-substitution. At each step, it substitutes implied values into the next type specifier and performs Boolean logic reduction. Abelson *et al.* [AS96] discuss Lisp algorithms for performing algebraic reduction; however, in addition to the Abelson algorithm, reducing Boolean expressions representing Common Lisp types involves additional reductions representing the subtype relations of terms in question. For example `(and number fixnum ...)` reduces to `(and fixnum ...)` because `fixnum` is a subtype of `number`. Similarly, `(or number fixnum ...)` reduces to `(or number ...)`. Newton *et al.* [NVC17] discuss techniques of Common Lisp type reduction in the presence of subtypes. Example 11.6 shows step-by-step reduction starting at the assumption that `(not (and number (not float)))` is `nil`, and ending at `(and float string (not number))` reducing to `string`.

Example 11.6 (Example of forward substitution).

```
(not (and number (not float))) = nil
  ⟹ (and number (not float)) = t
                  ⟹ number = t      and
                (not float) = t
                  ⟹ float = nil
(or float string (not number)) = (or nil string (not t))
                               = (or nil string nil)
                               = string
```

With this forward substitution, `reduced-typecase` is able to rewrite the second clause `((or float string (not number)) body-forms-2...)` simply as `(string body-forms-2...)`. Thereafter, a similar forward substitution is made to transform the third clause from `(string body-forms-3...)` to `(nil body-forms-3...)`.

Example 11.5 illustrates a situation in which a type specifier in one of the clauses is reduced to `nil`. In such a case, we would like the compiler to issue warnings about finding unreachable code, and in fact it does (at least when tested with SBCL[2]) because the compiler finds `nil` as the type specifier. The clauses in Example 11.7 are identical to those in Example 11.5 and consequently, the expressions `body-forms-3...` in the third clause cannot be reached. Yet, contrary to Example 11.5, SBCL, AllegroCL[3], and CLISP[4] issue no warning at all that `body-forms-3...` is unreachable code.

Example 11.7 (Invocation of `typecase` with unreachable code).

```
(typecase object
   ((not (and number (not float))) body-forms-1...)
   ((or float string (not number)) body-forms-2...)
   (string body-forms-3...))
```

11.2.2 Order dependency

We now reconsider Examples 11.2 and 11.3. While the semantics are the same, there is an important distinction in practice. The first `typecase` contains mutually exclusive clauses, whereas the second one does not. *E.g.*, if the `(member 40 41 42)` check is moved before the `(eql 42)` check, then `(eql 42)` will never match, and the consequent code `body-forms-2...` will be unreachable.

For the order of the type specifiers given in Example 11.2, the types can be simplified, having no redundant type checks, as shown in Example 11.3. This phenomenon is both a consequence of the particular types in question and also the order in which they occur. As a contrasting example, consider the situation in Example 11.8 where the first two clauses of the `typecase` are reversed with respect to Example 11.2. In this case, knowing that `OBJECT` is not of type `(and (member 40 41 42) (not (eql 42)))` tells us nothing about whether `OBJECT` is of type `(eql 42)`; so no reduction can be inferred.

Example 11.8 (Re-ordering clauses sometimes enable reduction).

```
(typecase OBJECT
  ((and (member 40 41 42) (not (eql 42)))
   body-forms-2...)
  ((eql 42)
   body-forms-1...)
  ((and fixnum (not (member 40 41 42)))
```

[2]We tested with SBCL 1.3.14. SBCL is an implementation of ANSI Common Lisp. `http://www.sbcl.org/`

[3]We tested with the International Allegro CL Free Express Edition, version 10.1 [32-bit Mac OS X (Intel)] (Sep 18, 2017 13:53). `http://franz.com`

[4]We tested with GNU CLISP 2.49, (2010-07-07). `http://clisp.cons.org/`

```
   body-forms-3...)
  ((and number (not fixnum))
   body-forms-4...))
```

Programmatic reductions in the `typecase` are dependent on the order of the specified types. There are many possible approaches to reducing types despite the order in which they are specified. We consider two such approaches. Section 11.2.5 discusses automatic reordering of disjoint clauses, and Section 11.3 uses decision diagrams.

As already suggested, a situation as shown in Example 11.8 can be solved to avoid the redundant type check, `(eql 42)`, by reordering the disjoint clauses as in Example 11.2. However, there are situations in which no reordering alleviates the problem. Consider the code shown in Example 11.9. We see that some sets of types are reorderable, allowing reduction, but for some sets of types such reordering is impossible. We consider in Section 11.3 `typecase` optimization where reordering is futile. For now we concentrate on efficient reordering wherever possible.

Example 11.9 (Re-ordering cannot always enable reduction)**.**

```
(typecase OBJECT
  ((and unsigned-byte (not bignum))
   body-forms-1...)
  ((and bignum (not unsigned-byte))
   body-forms-2...))
```

11.2.3 Mutually disjoint clauses

As suggested in Section 11.2.2, to arbitrarily reorder the clauses, the types must be disjoint. It is straightforward to transform any `typecase` into another which preserves the semantics but for which the clauses are reorderable. Consider a `typecase` in a general form.

Example 11.10 shows a set of type checks equivalent to those in Example 11.1 but with redundant checks, making the clauses mutually exclusive, and thus reorderable.

Example 11.10 (typecase with mutually exclusive type checks)**.**

```
(typecase OBJECT
  (Type.1
   body-forms-1...)
  ((and Type.2
        (not Type.1))
   body-forms-2...)
  ((and Type.3
        (not (or Type.1 Type.2)))
   body-forms-3...))
  ...
  ((and Type.n
        (not (or Type.1 Type.2 ... Type.n-1)))
   body-forms-n...))
```

In order to make the clauses reorderable, we make them more complex which might seem to defeat the purpose of optimization. However, as we see in Section 11.2.5, the complexity can sometimes be removed after reordering, thus resulting in a set of type checks which is better than the original. We discuss what we mean by better in Section 11.2.4.

We proceed by first describing a way to judge which of two given orders is better, and with that comparison function, we can visit every permutation and choose the best.

One might also wonder why we suffer the pain of establishing heuristics and visiting all permutations of the mutually disjoint types in order to find the best order. One might ask, why not just put the clauses in the

best order to begin with. The reason is because in the general case, it is not possible to predict what the best order is. As is discussed in Section 11.4, ordering the Boolean variables to produce the smallest binary decision diagram is an coNP-Complete [Bry86] problem. The only solution in general is to visit every permutation. The problem of ordering a set of type tests for optimal reduction must also be at least coNP-Complete, because if we had a better solution, we would be able to solve the BDD coNP-Complete problem as a consequence.

11.2.4 Comparing heuristically

Given a set of disjoint, and thus reorderable clauses, we can now consider finding a good order. We can examine a type specifier, typically after having been reduced, and heuristically assign a cost. A high cost is assigned to a `satisfies` type, a medium cost to `AND`, `OR`, and `NOT` types which takes into account the cost of the types specified therein, a cost to `member` types proportional to their length, and a low cost to atomic names and `eql` type specifiers.

To estimate the relative goodness of two, given, semantically identical, `typecase` invocations, we can heuristically estimate the complexity of each by using a weighted sum of the costs of the individual clauses. The weight of the first clause is higher because the type specified therein will be always checked. Each type specifier thereafter will only be checked if all the preceding checks fail. Thus the heuristic weights assigned to subsequent checks is chosen successively smaller as each subsequent check has a smaller probability of being reached at run-time.

11.2.5 Reduction with automatic reordering

Now that we have a way to heuristically measure the complexity of a given invocation of `typecase` we can therewith compare two semantically equivalent invocations and choose the better one. If the number of clauses is small enough, we can visit all possible permutations. If the number of clauses is large, we can sample the space randomly for some specified amount of time or specified number of samples, and choose the best ordering we find.

We introduce the macro, `auto-permute-typecase`. It accepts the same arguments as `typecase` and expands to a `typecase` form. It does so by transforming the specified types into mutually disjoint types as explained in Section 11.2.3, then iterating through all permutations of the clauses. For each permutation of the clauses, it reduces the types, eliminating redundant checks where possible using forward-substitution as explained in Section 11.2.1, and assigns a cost heuristic to each permutation as explained in Section 11.2.4. The `auto-permute-typecase` macro then expands to the `typecase` form with the clauses in the order which minimizes the heuristic cost.

Example 11.11 shows an invocation and expansion of `auto-permute-typecase`. In this example `auto-permute-typecase` does a good job of eliminating redundant type checks.

Example 11.11 (Invocation and expansion of `auto-permute-typecase`).

```
(auto-permute-typecase object
  ((and unsigned-byte (not (eql 42)))
   body-forms-1...)
  ((eql 42)
   body-forms-2...)
  ((and number (not (eql 42)) (not fixnum))
   body-forms-3...)
  (fixnum
   body-forms-4...))

(typecase object
  ((eql 42) body-forms-2...)
  (unsigned-byte body-forms-1...)
  (fixnum body-forms-4...)
  (number body-forms-3...))
```

As mentioned earlier, a particular optimization can be made in the situation where the type checks in the `typecase` are exhaustive; in particular the final type check may be replaced with `t`/`otherwise`. Example 11.12 illustrates such an expansion in the case that the types are exhaustive. Notice that the final type test in the expansion is `t`.

Example 11.12 (Invocation and expansion of `auto-permute-typecase` with exhaustive type checks)**.**

```
(auto-permute-typecase object
  ((or bignum unsigned-byte) body-forms-1...)
  (string body-forms-2...)
  (fixnum body-forms-3...)
  ((or (not string) (not number)) body-forms-4...))

(typecase object
  (string body-forms-2...)
  ((or bignum unsigned-byte) body-forms-1...)
  (fixnum body-forms-3...)
  (t body-forms-4...))
```

11.3 Decision diagram approach

In Section 11.2.5 we looked at a technique for reducing `typecase` based solely on programmatic manipulation of type specifiers. Now we explore a different technique based on a data structure known as Reduced Ordered Binary Decision Diagram (ROBDD).

Example 11.9 illustrates that redundant type checks cannot always be reduced via reordering. Example 11.13 is, however, semantically equivalent to Example 11.9. Successfully mapping the code from of a `typecase` to an ROBDD will guarantee that redundant type checks are eliminated. In the following sections we automate this code transformation.

Example 11.13 (Suggested expansion of Example 11.9)**.**

```
(if (typep object 'unsigned-byte)
    (if (typep object 'bignum)
        nil
        (progn body-forms-1...))
    (if (typep object 'bignum)
        (progn body-forms-2...)
        nil))
```

The code in Example 11.13 also illustrates a concern of code size explosion. With the two type checks `(typep object 'unsigned-byte)` and `(typep object 'bignum)`, the code expands to 7 lines of code. If this code transform is done naïvely, the risk is that each `if`/`then`/`else` effectively doubles the code size. In such an undesirable case, a `typecase` having N unique type tests among its clauses, would expand to $2^{N+1} - 1$ lines of code, even if such code has many congruent code paths. The use of ROBDD related techniques allows us to limit the code size to something much more manageable. Some discussion of this is presented in Section 11.4.

ROBDDs (Section 11.3.1) represent the semantics of Boolean equations but do not maintain the original evaluation order encoded in the actual code. In this sense the reordering of the type checks, which is explicit and of combinatorial complexity in the previous approach, is automatic in this approach. A complication is that normally ROBDDs express Boolean functions, so the mapping from `typecase` to ROBDD is not immediate, as a `typecase` may contain arbitrary side-effecting expressions which are not restricted to Boolean expressions. We employ an encapsulation technique which allows the ROBDDs to operate opaquely on these problematic expressions (Section 11.3.1). Finally, we are able to serialize an arbitrary `typecase` invocation into an efficient `if`/`then`/`else` tree (Section 11.3.3).

ROBDDs inherently eliminate duplicate checks. However, ROBDDs cannot easily guarantee removing all unnecessary checks as that would involve visiting every possible ordering of the leaf level types involved.

11.3.1 An ROBDD compatible type specifier

An ROBDD is a data structure used for performing many types of operations related to Boolean algebra. When we use the term ROBDD, we mean, as the name implies, a decision diagram (directed cyclic graph, DAG) whose vertices represent Boolean tests and whose branches represent the consequent and alternative actions.

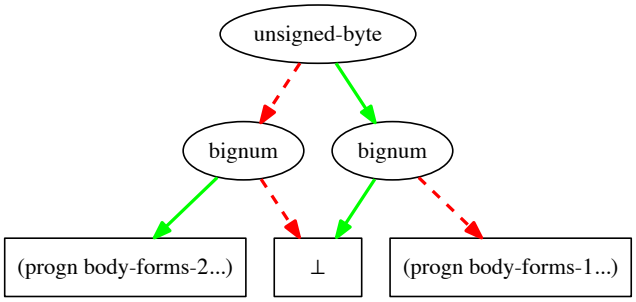


Figure 11.1: Decision Diagram representing irreducible `typecase`. This is similar to an ROBDD, but does not fulfill the definition thereof, because the leaf nodes are not simple Boolean values.

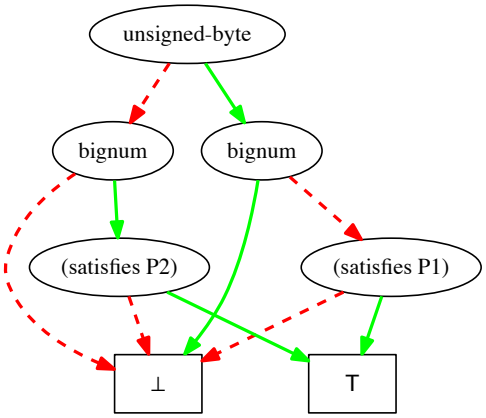


Figure 11.2: ROBDD with temporary valid `satisfies` types

An ROBDD has its variables **O**rdered, meaning that there is some ordering of the variables $\{v_1, v_2, ..., v_N\}$ such that whenever there is an arrow from v_i to v_j then $i < j$. An ROBDD is deterministically **R**educed so that all common sub-graphs are shared rather than duplicated. The reader is advised to read the lecture notes of Andersen [And99] for a detailed understanding of the reduction rules. It is worth noting that there is variation in the terminology used by different authors. For example, Knuth [Knu09] uses the unadorned term BDD for what we are calling an ROBDD.

A unique ROBDD is associated with a canonical form representing a Boolean function, or otherwise stated, with an equivalence class of expressions within the Boolean algebra. In particular, intersection, union, and complement operations as well as subset and equivalence calculations on elements from the underlying space of sets or types can be computed by straightforward algorithms. We omit detailed explanations of those algorithms here and, instead, we refer the reader to work by Andersen [And99] and Castagna [Cas16].

We employ ROBDDs to convert a `typecase` into an `if/then/else` diagram as shown in Figure 11.1. In the figure, we see a decision diagram which is similar to an ROBDD, at least in all the internal nodes of the diagram. Green arrows lead to the consequent if a specified type check succeeds. Red arrows lead to the alternative. However, the leaf nodes are not Boolean values as we expect for an ROBDD.

We want to transform the clauses of a `typecase` as shown in Example 11.1 into a binary decision diagram. To do so, we associate a distinct `satisfies` type with each clause of the `typecase`. Each such `satisfies` type has a unique function associated with it, such as `P1`, `P2`, etc, allowing us to represent the diagram shown in Figure 11.1 as an actual ROBDD as shown in Figure 11.2.

In order for certain Common Lisp functions to behave properly (such as `subtypep`) the functions `P1`, `P2`, etc. must be real functions, as opposed to place-holder functions types as Baker [Bak92] suggests, so that `(satisfies P1)` etc, have type specifier semantics. `P1`, `P2`, etc, must be defined in a way which preserves the semantics of the `typecase`.

Ideally we would like to create type specifiers such as the following:

```
(satisfies (lambda (object)
             (typep object '(and (not unsigned-byte)
                                 bignum))))
```

Unfortunately, the specification of `satisfies` explicitly forbids this, and requires that the operand of `satisfies` be a symbol representing a globally callable function, even if the type specifier is only used in a particular dynamic extent. Because of this limitation in Common Lisp, we create the type specifiers as follows. Given a type specifier, we create such a functions at run-time using the technique shown in the function `define-type-predicate` defined in Implementation 11.14, which programmatically defines function with semantics similar to those shown in Example 11.15.

Implementation 11.14 (define-type-predicate)**.**

```
(defun define-type-predicate (type-specifier)
  (let ((function-name (gensym "P")))
    (setf (symbol-function function-name)
          #'(lambda (object)
              (typep object type-specifier)))
    function-name))
```

Example 11.15 (Semantics of `satisfies` predicates)**.**

```
(defun P1 (object)
  (typep object '(and (not unsigned-byte) bignum)))
(defun P2 (object)
  (typep object '(and (not bignum) unsigned-byte)))
```

The `define-type-predicate` function returns the name of a named closure which the calling function can use to construct a type specifier. The name and function binding are generated in a way which has dynamic extent and is thus friendly with the garbage collector.

To generate the ROBDD shown in Figure 11.2 we must construct a type specifier equivalent to the entire invocation of `typecase`. From the code in Example 11.1 we have to assemble a type specifier such as in Example 11.16. This example is provided simply to illustrate the pattern of such a type specifier.

Example 11.16 (Type specifier equivalent to Example 11.1)**.**

```
(let ((P1 (define-type-predicate 'Type.1))
      (P2 (define-type-predicate
           '(and Type.2 (not Type.1))))
      (P3 (define-type-predicate
           '(and Type.3 (not (or Type.1 Type.2)))))
      ...
      (Pn (define-type-predicate
           '(and Type.n (not (or Type.1 Type.2
                                 ... Type.n-1))))))
   `(or (and Type.1
             (satisfies ,P1))
        (and Type.2
             (not Type.1)
             (satisfies ,P2))
        (and Type.3
             (not (or Type.1 Type.2))
             (satisfies ,P3))
        ...
        (and Type.n
```

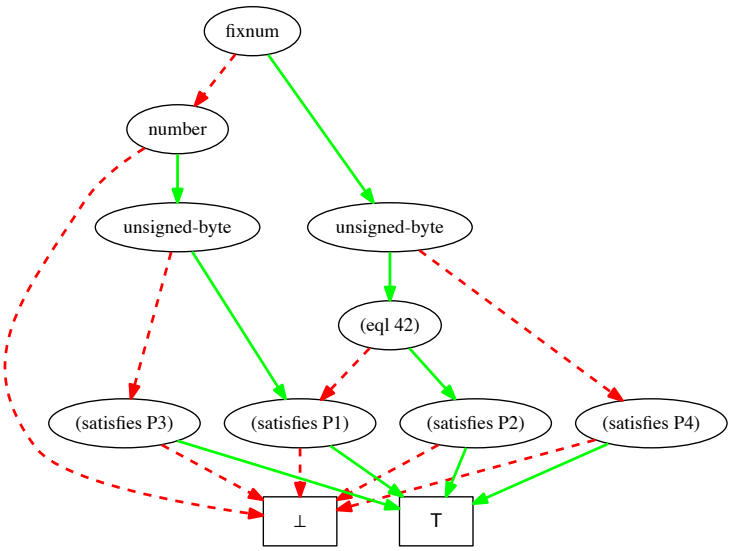


Figure 11.3: ROBDD generated from `typecase` clauses in Example 11.17

```
(not (or Type.1 Type.2
         ... Type.n-1))
(satisfies ,Pn))))
```

11.3.2 BDD construction from type specifier

Functions which construct an ROBDD need to understand a complete, deterministic ordering of the set of type specifiers via a compare function. To maintain semantic correctness the corresponding compare function must be deterministic. It would be ideal if the functions were able to give high priority to type specifiers which are likely to be seen at run time. We might consider, for example, taking clues from the order specified in the `typecase` clauses. We do not attempt to implement such decision making. Rather we choose to give high priority to type specifiers which are easy to check at run-time, even if they are less likely to occur.

We use a heuristic similar to that mentioned in Section 11.2.4 except that type specifiers involving `AND`, `OR`, and `NOT` never occur, rather such types correspond to algebraic operations among the ROBDDs themselves such that only non-algebraic types remain. More precisely, the heuristic we use is that atomic types such as `number` are considered fast to check, and `satisfies` types are considered slow. We recognize the limitation that the user might have used `deftype` to define a type whose name is an atom, but which is slow to type check. Ideally, we should fully expand user defined types into Common Lisp types. Unfortunately this is not possible in a portable way, and we make no attempts to implement such expansion in implementation specific ways. It is not even clear whether the various Common Lisp implementations have public APIs for the necessary operations.

A crucial exception in our heuristic estimation algorithm is that, to maintain the correctness of our technique, we must assure that the `satisfies` predicates emanating from `define-type-predicate` have the lowest possible priority. *I.e.*, as is shown in Figure 11.2, we must avoid that any type check appear below such a `satisfies` type in the ROBDD.

There are well known techniques for converting an ROBDD which represents a pure Boolean expression into an `if/then/else` expression which evaluates to `true` or `false`. However, in our case we are interested in more than simply the Boolean value. In particular, we require that the resulting expression evaluate to the same value as corresponding `typecase`. In Example 11.1, these are the values returned from `body-forms-1...`, `body-forms-2...`, ... `body-forms-n...`. In addition we want to assure that any side effects of those expressions are realized as well when appropriate, and never realized more than once.

We introduce the macro `bdd-typecase` which expands to a `typecase` form using the ROBDD technique. When the macro invocation in Example 11.17 is expanded, the list of `typecase` clauses is converted to a type specifier similar to what is illustrated in Example 11.16. That type specifier is used to create an ROBDD as illustrated in Figure 11.3. As shown in the figure, temporary `satisfies` type predicates are created corresponding

to the potentially side-effecting expressions `body-forms-1`, `body-forms-2`, `body-forms-3`, and `body-forms-4`. In reality these temporary predicates are named by machine generated symbols; however, in Figure 11.3 they are denoted `P1`, `P2`, `P3`, and `P4`.

Example 11.17 (Invocation of `bdd-typecase` with intersecting types)**.**

```
(bdd-typecase object
  ((and unsigned-byte (not (eql 42)))
   body-forms-1...)
  ((eql 42)
   body-forms-2...)
  ((and number (not (eql 42)) (not fixnum))
   body-forms-3...)
  (fixnum
   body-forms-4...))
```

11.3.3 Serializing the BDD into code

The macro `bdd-typecase` emits code as in Example 11.18, but may just as easily output code as in Example 11.19 based on `tagbody`/`go`. In both example expansions, we have replaced the more cryptic machine generated, uninterned symbols such as `#:l1070` and `#:|block1066|`, with more human-readable labels such as `L1` and `block-1`.

Example 11.18 (Macro expansion of Example 11.17 using `labels`)**.**

```
((lambda (object-1)
   (labels ((L1 () (if (typep object-1 'fixnum)
                       (L2)
                       (L7)))
            (L2 () (if (typep object-1 'unsigned-byte)
                       (L3)
                       (L6)))
            (L3 () (if (typep object-1 '(eql 42))
                       (L4)
                       (L5)))
            (L4 () body-forms-2...)
            (L5 () body-forms-1...)
            (L6 () body-forms-4...)
            (L7 () (if (typep object-1 'number)
                       (L8)
                       nil))
            (L8 () (if (typep object-1 'unsigned-byte)
                       (L5)
                       (L9)))
            (L9 () body-forms-3...))
     (L1)))
 object)
```

The `bdd-typecase` macro walks the ROBDD, such as the one illustrated in Figure 11.3, visiting each non-leaf node therein. Each node corresponding to a named closure type predicate is serialized as a tail call to the clauses from the `typecase`. Each node corresponding to a normal type test is serialized as left and right branches, either as a label and two calls to `go` as in Example 11.19, or a local function definition with two tail calls to other local functions as in Example 11.18.

Example 11.19 (Alternate expansion of Example 11.17 using `tagbody`/`go`)**.**

```
((lambda (object-1)
```

```
  (block block-1
    (tagbody
     L1 (if (typep object-1 'fixnum)
            (go L2)
            (go L7))
     L2 (if (typep object-1 'unsigned-byte)
            (go L3)
            (go L6))
     L3 (if (typep object-1 '(eql 42))
            (go L4)
            (go L5))
     L4 (return-from block-1
          (progn body-forms-2...))
     L5 (return-from block-1
          (progn body-forms-1...))
     L6 (return-from block-1
          (progn body-forms-4...))
     L7 (if (typep object-1 'number)
            (go L8)
            (return-from block-1 nil))
     L8 (if (typep object-1 'unsigned-byte)
            (go L5)
            (go L9))
     L9 (return-from block-1
          (progn body-forms-3...)))))
object)
```

11.3.4 Emitting compiler warnings

The ROBDD, as shown in Figure 11.3, can be used to generate the Common Lisp code semantically equivalent to the corresponding `typecase` as already explained in Section 11.3.3, but we can do even better. There are two situations where we might wish to emit warnings: (1) if certain code is unreachable, and (2) if the clauses are not exhaustive. Unfortunately, there is no standard way to incorporate these warnings into the standard compiler output. One might get tempted to simply emit a warning of type `style-warning` as is suggested by the `typecase` specification. However, this would be undesirable since there is no guarantee that the corresponding code was human-generated—ideally we would only like to see such style warnings corresponding to human generated code.

The list of unreachable clauses can be easily calculated as a function of which of the `P1`, `P2` ... predicates are missing from the serialized output. As seen in Figure 11.3, each of `body-forms-1`, `body-forms-2`, `body-forms-3`, and `body-forms-4` is represented as `P1`, `P2`, `P3`, and `P4`, so no such code is unreachable in this case.

We also see in Figure 11.3 that there is a path from the root node to the `nil` leaf node which does not pass through `P1`, `P2`, `P3`, or `P4`. This means that the original `typecase` is not exhaustive. The type of any such value can be calculated as the particular path leading to `nil`. In the case of Figure 11.3, `(and (not fixnum) (not number))`, which corresponds simply to `(not number)`, is such a type. *I.e.*, the original `bdd-typecase`, shown in Example 11.17, does not have a clause for non numbers.

11.4 Related work

In this chapter we examine a set of related optimizations of type simplification which seem to be missing from the SBCL [Rho08] compiler. Maclachan [Mac92] echoes the concern of Brooks and Gabriel [BG84] as to whether any Common Lisp compiler could ever implement all the optimizations which the Common Lisp committee claim "any good compiler" can take care. Brooks laments that it is unlikely that any such compiler will ever implement "a fraction of the tricks expected of it."

This chapter refers to the functions `make-bdd` and `bdd-cmp` without showing their implementations. The code is available via GitLab from the EPITA/LRDE public web page.[5] That repository contains several things. Most interesting for the context of BDDs is the Common Lisp package, `LISP-TYPES`.

[5] `https://gitlab.lrde.epita.fr/jnewton/regular-type-expression.git`, tagged version from 14 October 2018 is `version-1.1.4`.

As there are many individual styles of programming, and each programmer of Common Lisp adopts an individual and personal style, it is unknown how widespread the use of `typecase` is in practice, and consequently whether optimizing it is effort well spent. A casual look at the code in the current public Quicklisp[6] repository reveals a rule of thumb. 1 out of 100 files, and 1 out of 1000 lines of code use or make reference to `typecase`. When looking at the Common Lisp code of SBCL itself, we found about 1.6 uses of `typecase` per 1000 lines of code. We have made no attempt to determine which of the occurrences are comments, trivial uses, or test cases, and which ones are used in critical execution paths; however, we do loosely interpret these results to suggest that an optimized `typecase` either built into the `cl:typecase` or as an auxiliary macro may be of little use to most currently maintained projects. On the contrary, we suggest that having such an optimized `typecase` implementation may serve as motivation to some programmers to make use of it in new projects, at least in machine generated code such as explained by Newton *et al.* [NDV16, NV18b] explain. Since generic function dispatch conceptually bases branching choices on Boolean combinations of type checks, one naturally wonders whether our optimizations might be useful within the implementation of CLOS [KdRB91].

Newton *et al.* [NDV16, NV18b] present a mechanism to characterize the type of an arbitrary sequence in Common Lisp in terms of a rational language of the types of the sequence elements. The article explains how to build a finite state machine and, from that, construct Common Lisp code for recognizing such a sequence. The code associates the set of transitions existing from each state as a `typecase`. The article notes that such a machine generated `typecase` could greatly benefit from an optimizing `typecase`.

The `map-permutations` function (Section 11.2.5) works well for small lists, but requires a large amount of stack space to visit all the permutations of large lists. Knuth [Knu09] explores several iterative (not recursive) algorithms using various techniques, in particular by plain changes [Knu09, Algorithm P, page 42], by cyclic shifts [Knu09, Algorithm C, page 56], and by Erlich swaps [Knu09, Algorithm E, page 57]. A survey of these three algorithms can also be found in the Cadence SKILL Blog[7] which discusses an implementation in SKILL [Bar90], another Lisp dialect.

There is a large amount of literature about Binary Decision Diagrams of many varieties [Bry86, Bry92, Ake78, Col13, And99]. In particular Knuth [Knu09, Section 7.1.4] discusses worst-case and average sizes, which we alluded to in Section 11.3. Newton *et al.* [NVC17] discuss how the Reduced Ordered Binary Decision Diagram (ROBDD) can be used to manipulate type specifiers, especially in the presence of subtypes. Castagna [Cas16] discusses the use of ROBDDs (he calls them BDDs in that article) to perform type algebra in type systems which treat types as sets [HVP05, CL17, Ans94].

Common Lisp does not provide explicit pattern-matching [Aug85] capabilities, although several systems have been proposed such as Optima[8] and Trivia[9].

Decision tree techniques are useful in the efficient compilation of pattern-matching constructs in functional languages [Mar08]. An important concern in pattern-matching compilation is finding the best ordering of the variables which is known to be coNP-Complete [Bry86]. However, when using BDDs to represent type specifiers, we obtain representation (pointer) equality—simply by using a consistent ordering; finding the *best* ordering is not necessary for our application.

In Section 11.2.1 we mentioned the problem of symbolic algebraic manipulation and simplification. Ableson *et al.* [AS96, Section 2.4.3] discuss this with an implementation of rational polynomials. Norvig [Nor92, Chapter 8] discusses this in a use case of a symbolic mathematics simplification program. Both the Ableson and Norvig studies explicitly target a Lisp-literate audience.

11.5 Conclusion and perspectives

As illustrated in Example 11.11, the exhaustive search approach used in the `auto-permute-typecase` (Section 11.2.5) can often do a good job removing redundant type checks occurring in a `typecase` invocation. Unfortunately, as shown in Example 11.9, sometimes such optimization is algebraically impossible because of particular type interdependencies. In addition, an exhaustive search becomes unreasonable when the number of clauses is large. In particular there are $N!$ ways to order N clauses. This means there are $7! = 5040$ orderings of 7 clauses and $10! = 3,628,800$ orderings of 10 clauses.

On the other hand, the `bdd-typecase` macro, using the ROBDD approach (Section 11.3.2), is always able to remove duplicate checks, guaranteeing that no type check is performed twice. Nevertheless, it may fail to eliminate some unnecessary checks which need not be performed at all.

It is known that the size and shape of a reduced BDD depend on the ordering chosen for the variables [Bry86]. Furthermore, it is known that finding the best ordering is coNP-Complete [Bry86], and in this chapter we do not address questions of choosing or improving variable orderings. It would be feasible, at least in some cases,

[6] `https://www.quicklisp.org/`
[7] `https://community.cadence.com/tags/Team-SKILL`, SKILL for the Skilled, *Visiting all Permutations*
[8] `https://github.com/m2ym/optima` as of 14 October 2018.
[9] `https://github.com/guicho271828/trivia` as of 14 October 2018.

to apply the exhaustive search approach with ROBDDs. *I.e.*, we could visit all orders of the type checks to find which gives the smallest ROBDD. In situations where the number of different type tests is large, the development described in Section 11.3.1 might very well be improved employing some known techniques for improving BDD size though variable ordering choices [ATB94]. In particular, we might attempt to use the order specified in the `typecase` as input to the sorting function, attempting, at least in the simple cases, to respect the user given order as much as possible.

In Section 11.2.3, we presented an approach to approximating the cost of a set of type tests and commented that the heuristics are simplistic. We leave it as a matter for future research, how to construct good heuristics, which take into account the computational intensity of such manipulation.

We believe this research may be useful for two target audiences: application programmers and compiler developers. Even though the observed use frequency of `typecase` seems low in the majority of currently supported applications, programmers may find the macros explained in this report (`auto-permute-typecase` and `bdd-typecase`) to be useful in rare optimization cases, but more often for their ability to detect certain dubious code paths. There are, however, limitations to the portable implementation, namely the lack of a portable expander for user defined types and inability to distinguish between machine generated and human generated code. These shortcomings may not be significant limitations to the compiler implementer, in which case the compiler may be able to better optimize user types, implement better heuristics in terms of costs of certain type checks, and emit useful warnings about unreachable code.

Chapter 12

Conclusion

12.1 Contributions

The important contributions in this work are:

- Implementation of rational type expressions to efficiently recognize regular type patterns in Common Lisp sequences.
- The extension of ROBDDs in Chapter 7 to accommodate the Common Lisp type system.
- Numerical analysis of ROBDD worst case sizes including introduction of residual compression ratio.
- Algorithm for generating a worst-case ROBDD of n Boolean variables.
- Techniques for generating randomly selected ROBDD of n Boolean variables.
- Optimization of Common Lisp `typecase` to eliminate redundant type checks.
- Release of several Common Lisp packages to the community.[1]
- Publication of *Type-Checking of Heterogeneous Sequences in Common Lisp* in proceedings of the 2016 European Lisp Symposium.
- Publication of *Programmatic Manipulation of Common Lisp Type Specifiers* in the proceedings of the 2017 European Lisp Symposium.
- Publication of *Strategies for Typecase Optimization*, in the proceedings of the 2018 European Lisp Symposium.
- Publication of *Recognizing Hetergeneous Sequences by Rational Type Expression*, in the proceedings of the SPLASH-2018, Meta'18: Workshop on Meta-Programming Techniques and Reflection
- Publication of *A Theoretical and Numerical Analysis of the Worst-Case Size of Reduced Ordered Binary Decision Diagrams* in ACM journal *Transactions on Computational Logic*–publication date not yet known.

The following articles were published during the course of this research:

- Publication of *Type-Checking of Heterogeneous Sequences in Common Lisp* in proceedings of the 2016 European Lisp Symposium.
- Publication of *Programmatic Manipulation of Common Lisp Type Specifiers* in the proceedings of the 2017 European Lisp Symposium.
- Publication of *Strategies for Typecase Optimization*, in the proceedings of the 2018 European Lisp Symposium.
- Publication of *Recognizing Hetergeneous Sequences by Rational Type Expression*, in the proceedings of the SPLASH-2018, Meta'18: Workshop on Meta-Programming Techniques and Reflection
- Publication of *A Theoretical and Numerical Analysis of the Worst-Case Size of Reduced Ordered Binary Decision Diagrams* in ACM journal *Transactions on Computational Logic*–publication date not yet known.

[1] The source code contained in this report and that used during the research may be found and downloaded from the LRDE GitLab at `https://gitlab.lrde.epita.fr/jnewton/regular-type-expression` (tagged version from 14 October 2018 is `version-1.1.4`).

12.2 Perspective

As is the case with most research work, some open issues remain. We outline some of these issues here.

12.2.1 Common Lisp

For a better historical perspective, we would like to continue the investigation of the origins of the Common Lisp type system. We see several ideas in the Common Lisp type system which were areas of research outside the Lisp community during the period when Common Lisp was being developed. However, we have not been able to find citations between the two research communities. For instance Common Lisp defines type as a set of objects; similarly Hosoya and Pierce [Hos00, HP01] simplify their model of semantic subtyping by modeling a type as a set of values of the language.

12.2.2 Heterogeneous sequences

It is not clear whether Common Lisp could provide a way for a type definition in an application program to extend the behavior of `subtypep`. Having such a capability would allow such an extension for `rte`. Rational language theory does provide a well defined algorithm for deciding such questions given the relevant rational expressions [HMU06, Sections 4.1.1, 4.2.1]. It seems from the specification that a Common Lisp implementation is forbidden from allowing self-referential types, even in cases where it would be possible to do so.

For future extensions to this research we would like to experiment with extending the `subtypep` implementation to allow application level extensions, and therewith examine run-time performance when using `rte` based declarations within function definitions.

We would like continue the research into whether the core of this algorithm can be implemented in other dynamic languages, and to understand more precisely which features such a language would need to possess to support such implementation.

Several open questions remain:

Can regular type expressions be extended to implement more things we'd expect from a regular expression library? For example, could grouping somehow remember what was matched, and use that for `regexp-search-and-replace`? Additionally, would such a search and replace capability be useful?

Can the theory leading to regular type expressions be extended to tackle unification in a way which adds some sort of value?

One general problem with regular expressions is that if it is found that a sequence fails to match a given pattern, then we may like to know why it failed. Questions such as "How far did it match?" or "Where did it fail to match?" would be nice to answer. It is currently unclear whether the `rte` implementation can at all be extended to support such debugging features.

12.2.3 Binary decision diagrams

There are several obvious shortcomings in our intuitive evaluation of statistical variations in ROBDD sizes as discussed in Section 6.1.2. For example, we stated that judging from the small sample in Figure 6.8, it would appear that for large values of n, $|ROBDD_n|$ is a good estimate for average size. We would like to continue this investigation to better justify this gross approximation.

When using ROBDDs, or presumably 0-Sup-BDDs, one must use a hash table of all the BDDs encountered so far (or at least within a particular dynamic extent). This hash table, mentioned in Section 5.1, is used to ensure structural identity. However, it can become extremely large, even if its lifetime is short. Section 6.1 discusses the characterization of the worst-case size of an ROBDD as a function of the number of Boolean variables. This characterization ignores the transient size of the hash table, so one might argue that the size estimations in 6.1 are misleading in practice. More experimentation and analysis is needed to measure or estimate the hash table size, and to decrease the burden incurred. For example, we suspect that most of the hash table entries are never re-used.

As discussed in Section 6.3, Minato [Min93] claims that using the BDD variant called 0-Sup-BDD is well suited for sparse Boolean equations. We see potential applications for this claim in type calculations, especially when types are viewed as sets, as in Common Lisp. In such cases, the number of types is large, but each type constraint equation scantly concerns few types. Further experimentation is needed into 0-Sup-BDD based implementations of our algorithms, and contrast the performance results with those found thus far.

It is known that algorithms using BDDs achieve speed by allocating huge amounts of space. A question naturally arises: can we implement a fully functional BDD which never stores calculated values. The memory footprint of such an implementation would potentially be smaller, while incremental operations would be slower. It is not clear whether the overall performance would be better or worse. Castagna [Cas16] suggests a lazy version of the BDD data structure which may reduce the memory footprint, which would have a positive effect

on the BDD based algorithms. This approach suggests dispensing with the excessive heap allocation necessary to implement Andersen's approach [And99]. Moreover, our implementation (based on the Andersen model) contains additional debug features which increase the memory footprint. We would like to investigate which of these two approaches gives better performance, or allows us to solve certain problems. It seems desirable to attain heuristics to describe situations in which one or the other optimization approach is preferable.

Even though both Andersen [And99] and Minato [Min93] claim the necessity of enforcing structural identity, it is not clear whether, in our case, the run time cost associated with this memory burden outweighs the advantage gained by structural identity. Furthermore, the approach used by Castagna [Cas16] seems to favor laziness over caching, lending credence to our suspicion.

CUDD [Som] uses a common base data structure, DdNode, to implement several different flavors of BDD, including Algebraic Decision Diagrams (ADDs) and ZDDs. We have already acknowledged the need to experiment with other BDD flavors to efficiently represent run-time type based decisions such as the Common Lisp run-time type reflection [NVC17, NV18c] in performing simplification of type-related logic at compile-time. We wish to examine the question of whether the Common Lisp run-time type reflection can be improved by searching for better ordering of the type specifiers at compile-time. The work of Lozhkin [LS10] and Shannon [Sha49] may give insight into how much improvement is possible, and hence whether it is worth dedicating compilation time to it.

In Section 6.1.4, we examine the question of determining a sufficient sample size. The Engineering Statistics Handbook [Nat10] presents the work of Chakravart *et al.* [Kar68, pp 392-394] who in turn explain the Kolmogorov-Smirnov goodness of fit test. The test is designed to determine whether a sample in question comes from a specific distribution. We would like to apply this test to the sequence of samples in Figures 6.18 through 6.22 to assign a quantitative confidence to the histograms. This is a matter for further research.

12.2.4 Extending BDDs to accommodate Common Lisp types

As explained in Chapter 7, there is a remaining area of research which is of concern. There are exotic cases where two differently structured BDDs may represent the same Common Lisp type, particularly the empty type which may take many forms. We are not sure if this problem can be fixed, or whether it is just a weakness in an ambitious theory. For a comprehensive theory, we should completely characterize these situations. Our current recommendation is that the "final" types of any computation still need to be tested using a call to `subtypep`, lest they be subtypes of the `nil`.

We suspect the problem is that if `E` is a subtype of any subtree in the BDD, then we are safe, but if `E` is a subtype of some non-expressed combination, their uniqueness is not guaranteed.

12.2.5 MDTD

The results of the performance testing in Chapter 10 lead us to believe that the BDD as data structure for representing Common Lisp type specifiers is promising. However, there is still work to do, especially in identifying heuristics to predict its performance relative to more traditional approaches.

A shortcoming of our research is the lack of formal proofs of the correctness of our algorithms, most notably the graph decomposition algorithm from Section 9.4.

It has also been observed that, in the algorithm explained in Section 9.4, the convergence rate varies depending on the order of the reduction operations. We do not yet have sufficient data to characterize this dependence. Furthermore, the order of breaking the connections in the algorithm in Section 9.4 may be important. It is clear that many different strategies are possible, *e.g.*, (1) break busiest connections first, (2) break connections with the fewest dependencies, (3) random order, (4) closest to top of tree, etc. These are all areas of ongoing research.

We plan to investigate whether there are other applications of MDTD outside the Common Lisp type system. We hope anyone using Castagna's techniques [Cas16] on type systems with semantic subtyping may benefit from the optimizations we have discussed.

A potential application with Common Lisp is improving the `subtypep` implementation itself, which is known to be slow in some cases. Implementation 9.6 gave a BDD specific implementation of `bdd-subtypep`. We intend to investigate whether existing Common Lisp implementations could use our technique to represent type specifiers in their inferencing engines, and thereby make some subtype checks more efficient.

12.2.6 Optimizing `typecase`

It is known that the size and shape of a reduced BDD depend on the ordering chosen for the variables [Bry86]. Furthermore, it is known that finding the best ordering is coNP-Complete [Bry86], and in this chapter we do not address questions of choosing or improving variable orderings. It would be feasible, at least in some cases, to apply the exhaustive search approach with ROBDDs. *I.e.*, we could visit all orders of the type checks

to find which gives the smallest ROBDD. In situations where the number of different type tests is large, the development described in Section 11.3.1 might very well be improved employing some known techniques for improving BDD size though variable ordering choices [ATB94]. In particular, we might attempt to use the order specified in the `typecase` as input to the sorting function, attempting, at least in the simple cases, to respect the user-given order as much as possible.

12.2.7 Emergent issues

We know that PCL (Portable Common Loops) [BKK+86], which is the heart of many implementations of the CLOS (the Common Lisp Object System) [Kee89, Ano87] in particular the implementation within SBCL, uses decision trees in the form of discrimination nets to optimize generic function dispatch. Whether these discrimination nets can be replaced or simplified using ROBDDs is unknown.

In this research we have looked at BDDs as a source of canonicalizing Boolean expressions representing types. As we pointed out in Section 12.2.4, certain exotic cases are problematic. There are other normal forms which might be considered: Another mechanism which may reportedly be used for Boolean formula canonicalization is the Algebraic Normal Form (ANF) or Reed-Muller expansion [XAMM93].

We discovered very late in this project that a bottom-up approach to convert a Common Lisp type specifier to DNF form was in many cases much better performing than the fixed point approach discussed in Section 2.6. The code for this approach, `type-to-dnf-bottom-up` is also shown in Appendix A after the code for `type-to-dnf`. Without this alternative bottom-up approach we found that some of our test cases never finished in Allegro CL, and with this additional optimization the Allegro CL performance was very competitive with SBCL. On the other hand, we found that some of the performance tests in SBCL (as discussed in Chapter 10) exhausted the memory and we landed in the LDB (low level SBCL debugger). We believe there is merit to developing a correct, portable, high performance type simplifier; however this is a matter where more research and further experimentation is needed.

Appendix A

Code for `reduce-lisp-type`

The following Common Lisp code implements the algorithm explained in Section 2.6.

A.1 Fixed-point based type specifier simplification

Below is the implementation of the function `type-to-dnf` which is discussed in Section 2.6. The function is referred to by the name `reduce-lisp-type` in Section 2.6.

```
(defun type-to-dnf (type)
  (declare (optimize (speed 3) (debug 0) (compilation-speed 0) (space 0)))
  (labels ((and? (obj)
             (and (consp obj)
                  (eq 'and (car obj))))
           (or? (obj)
             (and (consp obj)
                  (eq 'or (car obj))))
           (not? (obj)
             (and (consp obj)
                  (eq 'not (car obj))))
           (flatten (f type)
             ;; (X a (X b c) d) --> (X a b c d)
             (cons (car type) (mapcan (lambda (t1)
                                        (if (funcall f t1)
                                            (copy-list (cdr t1))
                                            (list t1))) (cdr type))))
           (again (type)
             (cond
               ((atom type) type)
               ((member (car type) '(or and no))
                (cons (car type)
                      (mapcar #'type-to-dnf (cdr type))))
               (t
                type)))
           (to-dnf (type)
             (declare (type (or list symbol) type))
             (when (and? type)
               (when *reduce-member-type*
                 (setf type (reduce-member-type type))))
             (when (and? type)
               ;; (and a b NIL c d) --> NIL
               (when (member nil (cdr type) :test #'eq)
                 (setf type nil)))
             (when (and? type)
               ;; (and a b nil c d) --> (and a b c d)
               (when (member t (cdr type) :test #'eq)
                 (setf type (cons 'and (remove t (cdr type) :test #'eq)))))
             (when (and? type)
               ;; (and) --> t
               (when (null (cdr type))
                 (setf type t)))
```

```
(when (and? type)
  ;; (and a (and B C) d) --> (and a B C d)
  (while (some #'and? (cdr type))
    (setf type (flatten #'and? type))) )
(when (and? type)
  ;; (and a (or B C) d) -- (or (and a B d) (and a C d))
  (let ((hit (find-if #'or? (cdr type))))
    (when hit
      (setf type
            (cons 'or
                  (mapcar (lambda (t1)
                            (cons 'and (mapcar (lambda (t2)
                                                 (if (eq t2 hit)
                                                     t1
                                                     t2))
                                               (cdr type))))
                          (cdr hit)))))))
(when (and? type)
  ;; (and a b (not b) c) --> nil
  (when (exists t1 (cdr type)
          (and (not? t1)
               (member (cadr t1) (cdr type) :test #'equal)))
    (setf type nil)))
(when (and? type)
  (setf type (cons 'and (remove-duplicates (cdr type) :test #'equal))))
(when (and? type)
  ;; (and a) --> a
  (when (and (cdr type)
             (null (cddr type)))
    (setf type (cadr type))))
(when (or? type)
  ;; (or a b T c d) --> T
  (when (member t (cdr type) :test #'eq)
    (setf type t)))
(when (or? type)
  ;; (or a b nil c d) --> (or a b c d)
  (when (member nil (cdr type) :test #'eq)
    (setf type (cons 'or (remove nil (cdr type) :test #'eq)))))
(when (or? type)
  ;; (or) --> nil
  (when (null (cdr type))
    (setf type nil)))
(when (or? type)
  ;; (or a (or b c) d) --> (or a b c d)
  (while (some #'or? (cdr type))
    (setf type (flatten #'or? type))))
(when (or? type)
  ;; (or a b (not b) c) --> t
  (when (exists t1 (cdr type)
          (and (not? t1)
               (member (cadr t1) (cdr type) :test #'equal)))
    (setf type t)))
(when (or? type)
  (setf type (cons 'or (remove-duplicates (cdr type) :test #'equal))))
(when (or? type)
  ;; (or a) --> a
  (when (and (cdr type)
             (null (cddr type)))
    (setf type (cadr type))))
(when (equal type '(not t))
  ;; (not t) --> nil
  (setf type nil))
(when (equal type '(not nil))
  ;; (not nil) --> t
  (setf type t))
```

```
      (when (not? type)
        ;; (not (not a)) --> a
        (while (and (not? type)
                    (not? (cadr type)))
          (setf type (cadr (cadr type)))))
      (when (not? type)
        ;; (not (and a b)) --> (or (not a) (not b))
        (when (and? (cadr type))
          (setf type (cons 'or (mapcar (lambda (t1)
                                         (list 'not t1)) (cdr (cadr type)))))))
      (when (not? type)
        ;; (not (or a b))  --> (and (not a) (not b))
        (when (or? (cadr type))
          (setf type (cons 'and (mapcar (lambda (t1)
                                          (list 'not t1)) (cdr (cadr type)))))))
      (again type)))
  (fixed-point #'to-dnf
               (alphabetize-type type)
               :test #'equal)))
```

A.2 Bottom-up, functional style, type specifier simplification

Below is the implementation of the function `type-to-dnf-bottom-up` which is discussed in Section 2.9.

```
(defun type-to-dnf-bottom-up (type)
  (labels ((and? (obj)
             (and (consp obj)
                  (eq 'and (car obj))))
           (or? (obj)
             (and (consp obj)
                  (eq 'or (car obj))))
           (not? (obj)
             (and (consp obj)
                  (eq 'not (car obj))))
           (un-duplicate (zero operands)
             (remove-duplicates (remove-duplicates (remove zero operands)
                                                   :test #'eq)
                                :test #'equal))
           (make-op (op one zero operands
                     &aux (dup-free (un-duplicate zero operands)))
             (cond
               ((null dup-free)
                zero)
               ((null (cdr dup-free))
                (car dup-free))
               ((or (exists x dup-free
                      (and (not? x)
                           (member (cadr x) dup-free :test #'eq)))
                    (exists x dup-free
                      (and (not? x)
                           (member (cadr x) dup-free :test #'equal))))
                one)
               (t
                (cons op dup-free))))
           (make-or (operands)
             (make-op 'or
                      t ; one
                      nil ; zero
                      (remove-subs operands)))
           (make-and (operands)
             (if (exists t2 operands
                   (exists t1 operands
                     (and (not (eq t1 t2))
                          (cached-subtypep t1 (list 'not t2)))))
                 nil
```

```
      (make-op 'and
               nil ; one
               t ; zero
               (remove-supers operands))))
(do-and (a b)
  (cond ((eq a nil)
         nil)
        ((eq b nil)
         nil)
        ((eq a t)
         b)
        ((eq b t)
         a)
        ((and? a)
         (cond ((and? b)
                (make-and (append (cdr a) (cdr b))))
               ((or? b)
                (make-or (loop :for y :in (cdr b)
                               :collect (do-and a y))))
               (t
                (do-and a (list 'and b)))))
        ((or? a)
         (cond ((and? b)
                (make-or (loop :for x :in (cdr a)
                               :collect (do-and x b))))
               ((or? b)
                (make-or (loop :for x :in (cdr a)
                               :nconc (loop :for y :in (cdr b)
                                            :collect (do-and x y)))))
               (t
                (do-and a (list 'and b)))))
        (t
         (do-and (list 'and a) b))))
(do-or (a b)
  ;; The do-and and do-or functions are not duals, becasue
  ;; both arguments a and b are in dnf form.  This means
  ;; (or (and ...)...) is a valid value of a or b but
  ;; (and (or ...) ...) is not.
  (cond ((eq a t)
         t)
        ((eq b t)
         t)
        ((eq a nil)
         b)
        ((eq b nil)
         a)
        ((and? a)
         (cond ((or? b)
                (make-or (cons a (cdr b))))
               (t
                (make-or (list a b)))))
        ((or? a)
         (cond ((or? b)
                (make-or (append (cdr a) (cdr b))))
               (t
                (do-or a (list 'or b)))))
        (t
         (cond ((or? b)
                (do-or (list 'or a) b))
               (t
                (make-or (list a b)))))))
(do-not (a)
  (cond ((eq a t)
         nil)
        ((eq a nil)
         t)
```

```
                ((and? a)
                 (make-or (mapcar #'do-not (cdr a))))
                ((or? a)
                 (make-and (mapcar #'do-not (cdr a))))
                ((not? a)
                 (cadr a))
                (t
                 (list 'not a)))))
(cond
  ((and? type)
   (reduce #'do-and (mapcar #'type-to-dnf-bottom-up (cdr type))
           :initial-value t))
  ((or? type)
   (reduce #'do-or (mapcar #'type-to-dnf-bottom-up (cdr type))
           :initial-value nil))
  ((not? type)
   (do-not (type-to-dnf-bottom-up (cadr type))))
  (t
   type))))
```

Appendix B

Code for s-expression baseline algorithm

The following Common Lisp code implements the algorithm explained in Section 9.1. The function has been subject to some simple optimizations. Note, the local function `disjoint?` which uses a hash table, `known-intersecting`, to mitigate the cost of the $\mathcal{O}(n^2)$ search incurred by the location functions `remove-disjoint` and `find-intersecting`.

```
(defun mdtd-baseline (type-specifiers)
  ;; the list of disjoint type-specifiers
  (let ((known-intersecting (make-hash-table :test #'equal)) decomposition)
    (labels ((disjoint? (T1 T2 &aux (key (list T1 T2)))
               (multiple-value-bind (hit found?) (gethash key known-intersecting)
                 (cond
                   (found? hit)
                   (t (setf (gethash key known-intersecting)
                            (disjoint-types-p T1 T2))))))
             (remove-disjoint (&aux (disjoint (setof T1 type-specifiers
                                                (forall T2 type-specifiers
                                                  (or (eq T2 T1)
                                                      (disjoint? T1 T2))))))
               (setf type-specifiers (set-difference type-specifiers disjoint
                                                     :test #'eq))
               (setf decomposition (union decomposition
                                          (remove nil ;; don't remember the nil type
                                                  (mapcar #'reduce-lisp-type disjoint))
                                          :test #'equivalent-types-p)))
             (find-intersecting ()
               (mapl (lambda (T1-tail &aux (T1 (car T1-tail)) (tail (cdr T1-tail)))
                       (dolist (T2 tail)
                         (unless (disjoint? T1 T2)
                           (return-from find-intersecting (values t T1 T2)))))
                     type-specifiers)
               nil)
             (forget (type)
               (setf type-specifiers (remove type type-specifiers :test #'eq)))
             (remember (type)
               (pushnew type type-specifiers :test #'equivalent-types-p)))
      (while type-specifiers
        (remove-disjoint)
        (multiple-value-bind (foundp T1 T2) (find-intersecting)
          (when foundp
            (forget T1)
            (forget T2)
            (remember `(and ,T1 ,T2))
            (remember `(and ,T1 (not ,T2)))
            (remember `(and (not ,T1) ,T2)))))
      decomposition)))
```

Appendix C

Code for BDD baseline algorithm

The following Common Lisp code implements the algorithm explained in Section 9.5.1. The code uses several terse variable names. This is both for formatting reasons, so that lines do not extend to far to the right, and also to coincide well with the variable names in Algorithm 6.

U: The set of BDDs being treated. Initially this corresponds one to one with the given type specifiers, except that those representing the empty type have been removed.

V: The set of BDDs during the call to `REDUCE`. Also used as a return value of `REDUCE`. V represents the refined set of BDDs having some of the BDDs operated on by the intersection and relative complement operations.

A: Each call to `REDUCE` takes one *pivot* element, A.

B: Each iteration within the `REDUCE` treats one BDD, B.

AB: Each iteration within the `REDUCE` examines the intersection type $AB = A \cap B$.

D: The set BDDs representing disjoint types. When the function finishes, this represents the complete set of disjoint types. Note the call to `(mapcar #'bdd-to-dnf ...)` on the last line to calculate the type specifier s-expressions from the BDDs.

```
(defun mdtd-bdd (type-specifiers)
  (bdd-with-new-hash
   (lambda (&aux (U (remove-if #'bdd-empty-type
                               (mapcar #'bdd type-specifiers))))
     (labels ((try (U D &aux (A (car U)))
                (cond
                  ((null U)
                   D)
                  (t
                   (flet ((reduction (acc B
                                      &aux (AB (bdd-and A B)))
                            (destructuring-bind (all-disjoint? V) acc
                              (cond
                                ((bdd-empty-type AB)
                                 (list all-disjoint? (adjoin B V)))
                                (t
                                 (list nil
                                       (union (remove-duplicates
                                               (remove-if #'bdd-empty-type
                                                 (list AB
                                                       (bdd-and-not A AB)
                                                       (bdd-and-not B AB))))
                                              V)))))))
                     (destructuring-bind (all-disjoint? V)
                         (reduce #'reduction (cdr U) :initial-value '(t nil))
                       (try V
                            (if all-disjoint?
                                (pushnew A D)
                                D)))))))))
       (mapcar #'bdd-to-dnf (try U nil))))))
```

Appendix D

Code for graph-based algorithm

The following is the CLOS based Common Lisp code implementing `DecomposeByGraph-1` and `DecomposeByGraph-2` functions as discussed in Section 9.4. The code of the s-expression based version and BDD based versions are implemented in terms of CLOS sublasses in Sections D.11 and D.12.

D.1 Support code for graph decomposition

```
(defvar *node-num* 0)

(defclass node ()
  ((id :type unsigned-byte :reader id :initform (incf *node-num*))
   (label :initarg :label :accessor label)
   (touches :initform nil :type list :accessor touches)
   (subsets :initform nil :type list :accessor subsets)
   (supersets :initform nil :type list :accessor supersets)))

(defmethod print-object ((n node) stream)
  (print-unreadable-object (n stream :type t :identity nil)
    (format stream "~D: ~A" (id n) (label n))))

(defgeneric add-node (graph node))
(defgeneric node-and (node1 node2))
(defgeneric node-and-not (node1 node2))
(defgeneric node-subtypep (node1 node2))
(defgeneric node-empty-type (node))
(defgeneric node-disjoint-types-p (node1 node2))

(defclass graph ()
  ((nodes :type list :accessor nodes :initarg :nodes
          :initform nil)
   (blue :type list :accessor blue
         :initform nil)
   (green :type list :accessor green
          :initform nil)
   (disjoint :type list :accessor disjoint
             :initform nil)))

(defgeneric extract-disjoint (graph))
(defgeneric decompose-graph-1 (g))
(defgeneric decompose-graph-2 (g))
```

D.2 Entry Point Functions

Implementation of DECOMPOSEBYGRAPH-1 from Algorithm 8 and DECOMPOSEBYGRAPH-2 from Algorithm 9.

```
(defun decompose-by-graph-1 (u &key (graph-class 'sexp-graph))
  (declare (type list u))
  (decompose-graph-1 (construct-graph graph-class u)))

(defun decompose-by-graph-2 (u &key (graph-class 'sexp-graph))
```

```
  (declare (type list u))
  (decompose-graph-2 (construct-graph graph-class u)))

(defmethod decompose-graph-1 ((g graph))
  (loop :while (or (blue g) (green g))
        :do (dolist (x->y (blue g))
              (destructuring-bind (x y) x->y
                (break-relaxed-subset g x y)))
        :do (dolist (xy (green g))
              (destructuring-bind (x y) xy
                (break-touching g x y))))
  (extract-disjoint g))

(defmethod decompose-graph-2 ((g graph))
  (loop :while (or (blue g) (green g))
        :do (dolist (x->y (blue g))
              (destructuring-bind (x y) x->y
                (break-strict-subset g x y)))
        :do (dolist (xy (green g))
              (destructuring-bind (x y) xy
                (break-touching g x y)))
        :do (dolist (x->y (blue g))
              (destructuring-bind (x y) x->y
                (break-loop g x y))))
  (extract-disjoint g))
```

D.3 Graph Construction

Implementation of ConstructGraph from Algorithm 10.

```
(defun construct-graph (graph-class u)
  (declare (type list u))
  (let ((g (make-instance graph-class)))
    (dolist (label u)
      (add-node g label))
    (mapl (lambda (tail)
            (let ((x (car tail)))
              (mapc (lambda (y)
                      (cond
                        ((node-subtypep x y)
                         (add-blue-arrow g x y))
                        ((node-subtypep y x)
                         (add-blue-arrow g y x))
                        (t
                         (multiple-value-bind (disjoint trust)
                             (node-disjoint-types-p x y)
                           (cond
                             ((null trust)
                              ;; maybe intersection types, not sure
                              (add-green-line g x y))
                             (disjoint
                              nil)
                             (t ;; intersecting types
                              (add-green-line g x y)))))))
                    (cdr tail)))) (nodes g))
    (dolist (node (nodes g))
      (maybe-disjoint-node g node))
    g))
```

D.4 Disjoining Nodes

Implementation of MaybeDisjointNode from Algorithm 11.

```
(defun maybe-disjoint-node (g node)
```

```
  (declare (type graph g) (type node node))
  (cond
    ((node-empty-type node)
     (setf (nodes g) (remove node (nodes g) :test #'eq)))
    ((null (or (touches node)
               (supersets node)
               (subsets node)))
     (setf (nodes g) (remove node (nodes g) :test #'eq))
     (pushnew node (disjoint g) :test #'eq))))
```

D.5 Green Line functions

Implementation of AddGreenLine from Algorithm 12.

```
(defun sort-nodes (n1 n2)
  (declare (type node n1 n2))
  (if (< (id n1) (id n2))
      (list n1 n2)
      (list n2 n1)))

(defun add-green-line (g x y)
  (declare (type graph g) (type node x y))
  (pushnew (sort-nodes x y) (green g) :test #'equal)
  (pushnew x (touches y) :test #'eq)
  (pushnew y (touches x) :test #'eq))

(defun delete-green-line (g x y)
  (declare (type graph g) (type node x y))
  (setf (green g)   (remove (sort-nodes x y) (green g) :test #'equal)
        (touches y) (remove x (touches y) :test #'eq)
        (touches x) (remove y (touches x) :test #'eq))
  (maybe-disjoint-node g x)
  (maybe-disjoint-node g y))
```

D.6 Blue Arrow Functions

Implementation of AddBlueArrow from Algorithm 14.

```
(defun add-blue-arrow (g x y)
  (pushnew (list x y) (blue g) :test #'equal)
  (pushnew x (subsets y) :test #'eq)
  (pushnew y (supersets x) :test #'eq))

(defun delete-blue-arrow (g x y)
  (declare (type graph g) (type node x y))
  (setf (blue g)      (remove (list x y) (blue g) :test #'equal)
        (subsets y)   (remove x (subsets y) :test #'eq)
        (supersets x) (remove y (supersets x) :test #'eq))
  (maybe-disjoint-node g x)
  (maybe-disjoint-node g y))
```

D.7 Strict Subset

Implementation of BreakStrictSubset from Algorithm 16.

```
(defun break-strict-subset (g sub super)
  (declare (type graph g) (type node sub super))
  (cond
    ((null (member super (supersets sub) :test #'eq))
     nil)
    ((subsets sub)
     nil)
    ((touches sub)
```

```
    nil)
   (t
    (setf (label super) (node-and-not super sub))
    (delete-blue-arrow g sub super)))
  g)
```

D.8 Relaxed Subset

Implementation of BreakRelaxedSubset from Algorithm 17.

```
(defun break-relaxed-subset (g sub super)
  (declare (type graph g) (type node sub super))
  (cond ((null (member super (supersets sub) :test #'eq))
         nil)
        ((subsets sub)
         nil)
        (t
         (setf (label super) (node-and-not super sub))
         (dolist (alpha (intersection (touches sub) (subsets super)
                                      :test #'eq))
           (add-green-line g alpha super)
           (delete-blue-arrow g alpha super))
         (delete-blue-arrow g sub super)))
  g)
```

D.9 Touching Connections

Implementation of BreakTouching from Algorithm 18.

```
(defun break-touching (g x y)
  (declare (type graph g) (type node x y))
  (cond
    ((null (member y (touches x) :test #'eq))
     nil)
    ((subsets x)
     nil)
    ((subsets y)
     nil)
    (t
     (let ((z (add-node g (node-and x y))))
       (psetf (label x) (node-and-not x y)
              (label y) (node-and-not y x))
       (dolist (alpha (union (supersets x) (supersets y) :test #'eq))
         (add-blue-arrow g z alpha))
       (dolist (alpha (intersection (touches x) (touches y) :test #'eq))
         (add-green-line g z alpha))
       (maybe-disjoint-node g z))
     (delete-green-line g x y)))
  g)
```

D.10 Breaking the Loop

Implementation of BreakLoop from Algorithm 19.

```
(defun break-loop  (g x y)
  (declare (type graph g) (type node x y))
  (cond
    ((null (member y (touches x) :test #'eq))
     nil)
    ((subsets x)
     nil)
    ((subsets y)
     nil)
```

```
  (t
   (let ((z (add-node g (node-and x y))))
     (setf (label x) (node-and-not x y))
     (dolist (alpha (touches x))
       (add-green-line g z alpha))
     (dolist (alpha (union (supersets x) (supersets y) :test #'eq))
       (add-blue-arrow g z alpha))
     (add-blue-arrow g z y)
     (add-blue-arrow g z x)
     (delete-blue-arrow g x y))))
g)
```

D.11 S-expression based graph algorithm

The following is a CLOS Common Lisp implementation of the algorithm described in Section 9.4. The algorithm represents Common Lisp types as s-expressions which Common Lisp refers to as *type specifiers*.

```
(defclass sexp-node (node)
  ((label :type (or list symbol))))

(defmethod node-and-not ((x sexp-node) (y sexp-node))
  (reduce-lisp-type `(and ,(label x) (not ,(label y)))))

(defmethod node-and  ((x sexp-node) (y sexp-node))
  (reduce-lisp-type `(and ,(label x) ,(label y))))

(defmethod node-empty-type ((node sexp-node))
  (null (label node)))

(defmethod node-subtypep ((x sexp-node) (y sexp-node))
  (subtypep (label x) (label y)))

(defmethod node-disjoint-types-p ((x sexp-node) (y sexp-node))
  (disjoint-types-p (label x) (label y)))

(defclass sexp-graph (graph)
  ())

(defmethod add-node ((g sexp-graph) type-specifier)
  (let ((z (make-instance 'sexp-node :label type-specifier)))
    (push z
          (nodes g))
    z))

(defmethod extract-disjoint ((g sexp-graph))
  (mapcar #'label (disjoint g)))
```

D.12 BDD based graph algorithm

The following is a CLOS Common Lisp implementation of the algorithm described in Section 9.4. The algorithm represents Common Lisp types BDDs as discussed in Section 9.5.2.

```
(defclass node-of-bdd (node)
  ((label :type bdd)))

(defmethod node-and-not ((x node-of-bdd) (y node-of-bdd))
  (bdd-and-not (label x) (label y)))

(defmethod node-and ((x node-of-bdd) (y node-of-bdd))
  (bdd-and (label x) (label y)))

(defmethod node-empty-type ((node node-of-bdd))
  (eq *bdd-false* (label node)))

(defmethod node-subtypep ((x node-of-bdd) (y node-of-bdd))
  (bdd-subtypep (label x) (label y)))

(defmethod node-disjoint-types-p ((x node-of-bdd) (y node-of-bdd))
  (values (bdd-disjoint-types-p (label x) (label y))
          t))

(defclass bdd-graph (graph)
  ())

(defmethod add-node ((g bdd-graph) type-specifier)
  (let ((z (make-instance 'node-of-bdd :label (bdd type-specifier))))
    (push z
          (nodes g))
    z))

(defmethod extract-disjoint ((g bdd-graph))
  (mapcar #'bdd-to-dnf (mapcar #'label (disjoint g))))

(defmethod decompose-graph-1 ((g bdd-graph))
  (bdd-call-with-new-hash
   (lambda ()
     (call-next-method))))

(defmethod decompose-graph-2 ((g bdd-graph))
  (bdd-call-with-new-hash
   (lambda ()
     (call-next-method))))
```

Appendix E

Subclasses of `function` using Clos

In Common Lisp the user may create subtypes of `function`. As shown in Implementation E.1, this is useful in implementing objects which have states like Clos objects, but may also be called as functions, using `apply`, `funcall`, mapping functions etc.

Implementation E.1 (User defined subtype/subclass of `function`).

```
(defclass reflectable-function ()
  ((function :initarg :function :type function)
   (lambda-list :initarg :lambda-list
                :reader lambda-list
                :type (or symbol list)))
  (:metaclass funcallable-standard-class))

(defmethod initialize-instance
     :after ((self reflectable-function) &key)
  (closer-mop:set-funcallable-instance-function
     self (slot-value self 'function)))

(defmacro  (lambda-list &rest body)
  `(make-instance 'reflectable-function
                  :function (lambda ,lambda-list ,@body)
                  :lambda-list ',lambda-list))
```

We see in Example E.2 that the user defined type `reflectable-function` is a subtype of `function`, and behaves well with the `subtypep`.

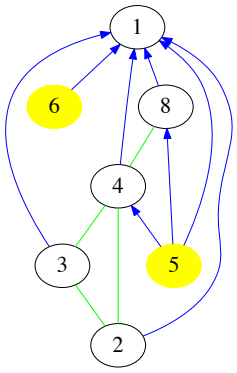
Example E.2 (Using an instance of user defined subtype of `function`).

```
CL-USER> (mapcar (reflecting-lambda (x) (list x)) '(a b c))
((A) (B) (C))
CL-USER> (type-of (reflecting-lambda (x) (list x)))
REFLECTABLE-FUNCTION
CL-USER> (subtypep 'reflectable-function 'function)
T
T
```

Appendix F

Running the graph-based algorithm on an example

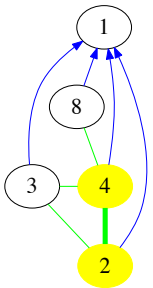
In this section we continue the disucssion began in Section 9.4.



Node	Boolean expression
1	A_1
2	A_2
3	A_3
4	A_4
5	A_5
6	A_6
7	A_7
8	A_8

Figure F.1: State 0: Topology graph

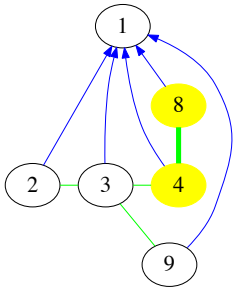
Nodes ⑤ and ⑥ in Figure F.1 meet the *strict subset* conditions, thus the arrow connecting them to their supersets, $5 \rightarrow 1$, $5 \rightarrow 4$, $5 \rightarrow 8$, and $6 \rightarrow 1$ can be eliminated and the superset nodes relabeled. *I.e.* ⑧ relabeled $A_8 \mapsto A_8 \cap \overline{A_5}$, ④ relabeled $A_4 \mapsto A_4 \cap \overline{A_5}$, and ① relabeled $A_1 \mapsto A_1 \cap \overline{A_5} \cap \overline{A_6}$. The result of these operations is that nodes ⑤ and ⑥ have now severed all connections, and are thus isolated. The updated graph is shown in Figure F.2.



Node	Boolean expression
1	$A_1 \cap \overline{A_5} \cap \overline{A_6}$
2	A_2
3	A_3
4	$A_4 \cap \overline{A_5}$
5	A_5
6	A_6
7	A_7
8	$A_8 \cap \overline{A_5}$

Figure F.2: State 1: Topology graph, after isolating 5 and 6.

The green line between nodes as ② and ④ in Figure F.2 meets the *touching connections* conditions. The nodes can thus be *separated* by breaking the connection, deleting the green line. To do this we must introduce a new node which represents the intersection of the sets ② and ④. The new node is labeled as the Boolean intersection: $A_2 \cap A_4 \cap \overline{A_5}$, and is labeled ⑨ in Figure F.3.



Node	Boolean expression
1	$A_1 \cap \overline{A_5} \cap \overline{A_6}$
2	$A_2 \cap \overline{A_4 \cap \overline{A_5}}$
3	A_3
4	$A_4 \cap \overline{A_5} \cap \overline{A_2}$
5	A_5
6	A_6
7	A_7
8	$A_8 \cap \overline{A_5}$
9	$A_2 \cap A_4 \cap \overline{A_5}$

Figure F.3: State 2: Topology graph after disconnecting 2 from 4

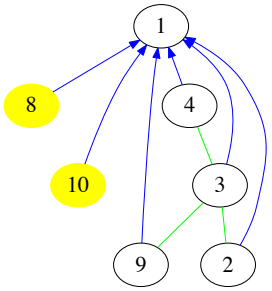
We must also introduce new blue lines from ⑨ to *any* node that either ② points to or ④ points to, which is ① in this case.

In addition we must draw green lines to nodes which both ② and ④ have green lines touching. In this cases that is only the node ③. So a green line is drawn between ⑨ and ③.

The green line between ② and ④ is deleted. The two nodes are relabeled: ②: $A_2 \mapsto A_2 \cap \overline{A_4 \cap \overline{A_5}}$ and ④: $A_4 \cap \overline{A_5} \mapsto A_4 \cap \overline{A_5} \cap \overline{A_2}$.

These graph operations should continue until all the nodes have become isolated. Observing Figure F.3 we see that several green lines meet the *touching connections*: ② — ③, ③ — ④, ③ — ⑨, and ④ — ⑧. It is not clear which of these connections should be broken next. *I.e.* what is the best strategy to employ when choosing the order to break connections. This is a matter for further research; we don't suggest any best strategy at this time. Nevertheless, we continue the segmentation algorithm a couple more steps.

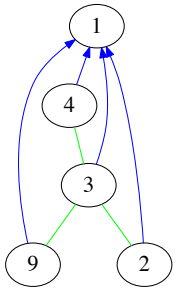
In Figure F.3, consider eliminating the green connection ④ — ⑧. We introduce a new node ⑩ representing the intersection, thus associated with the Boolean expression $A_4 \cap \overline{A_5} \cap \overline{A_2} \cap A_8 \cap \overline{A_5}$. The union of the supersets of ④ and ⑧, *i.e.* the union of the destinations of the arrows leaving ④ and ⑧ is just the node ①, thus we must introduce a blue arrow ⑩ → ①. There are no nodes which both ④ and ⑧ touch with a green line, so no green lines need to be added connecting to ⑩. We now relabel ④ and ⑧ with the respective relative complements. $8 \mapsto 8 \cap \overline{4}$ and $4 \mapsto 4 \cap \overline{8}$. The Boolean expressions are shown in Figure F.4.



Node	Boolean expression
1	$A_1 \cap \overline{A_5} \cap \overline{A_6}$
2	$A_2 \cap \overline{A_4 \cap \overline{A_5}}$
3	A_3
4	$A_4 \cap \overline{A_5} \cap \overline{A_2} \cap \overline{A_8 \cap \overline{A_5}}$
5	A_5
6	A_6
7	A_7
8	$A_8 \cap \overline{A_5} \cap \overline{A_4 \cap \overline{A_5} \cap \overline{A_2}}$
9	$A_2 \cap A_4 \cap \overline{A_5}$
10	$A_4 \cap \overline{A_5} \cap \overline{A_2} \cap A_8 \cap \overline{A_5}$

Figure F.4: State 3: Topology graph after disconnecting 4 from 8

Observing Figure F.4, we see it is possible to disconnect ⑧ from ① and thereafter disconnect ⑩ from ①. Actually you may choose to do this in either order. We will operate on ⑧ and then on ⑩, to result in the graph in Figure F.5.



Node	Boolean expression
1	$A_1 \cap \overline{A_6} \cap \overline{A_8 \cap \overline{A_5} \cap \overline{A_4 \cap \overline{A_5} \cap \overline{A_2}} \cap \overline{A_4 \cap \overline{A_5} \cap \overline{A_2} \cap A_8 \cap \overline{A_5}}}$
2	$A_2 \cap \overline{A_4 \cap \overline{A_5}}$
3	A_3
4	$A_4 \cap \overline{A_5} \cap \overline{A_2} \cap \overline{A_8 \cap \overline{A_5}}$
5	A_5
6	A_6
7	A_7
8	$A_8 \cap \overline{A_5} \cap \overline{A_4 \cap \overline{A_5} \cap \overline{A_2}}$
9	$A_2 \cap A_4 \cap \overline{A_5}$
10	$A_4 \cap \overline{A_5} \cap \overline{A_2} \cap A_8 \cap \overline{A_5}$

Figure F.5: State 4: Topology graph after isolating 8 and 10

From Figure F.5 it should be becoming clear that the complexity of the Boolean expressions in each node is becoming more complex. If we continue this procedure, eliminating all the blue arrows and green connecting lines, we will end up with 13 isolated nodes (each time a green line is eliminated one additional node is added). Thus the Boolean expressions can become exceedingly complex. A question which arises is whether it is better to *simplify* the Boolean expressions at each step, or whether it is better to wait until the end. The data structure shown in Section 9.5 promises to obviate that dilemma.

There are some subtle corner cases which may not be obvious. It is possible in these situations to end up with some disjoint subsets which are empty. It is also possible also that the same subset is derived by two different operations in the graph, but whose equations are very different.

This phenomenon is a result of a worst case assumption, *green intersection* in the algorithm. Consider a case where nodes Ⓐ, Ⓑ, and Ⓒ mutually connected with green lines signifying that the corresponding sets touch (are not disjoint). If the connection Ⓐ — Ⓑ is broken, a new green line must be drawn between the new node Ⓓ and Ⓒ. Why? Because it is possible that the set represented by $A \cap B \nparallel C$. However, this it is not guaranteed. It may very well be the case that both $A \nparallel C$ and $B \nparallel C$ while $A \cap B \parallel C$. Consider the simple example $A = \{1, 2\}$, $B = \{2, 3\}$, $C = \{1, 3\}$. $A \nparallel C$, $B \nparallel C$, but $A \cap B = \{2\} \parallel \{1, 3\} = C$.

This leads to the possibility that there be green lines in the topology graph which represent phantom connections. Later on in the algorithm when the green line between Ⓓ and Ⓒ is broken redundant sets may occur. Nodes Ⓒ and Ⓓ will be broken into three, $C \cap D$, $C \cap \overline{D}$, and $D \cap \overline{C}$. But $C \cap D = \varnothing$, $C = C \cap \overline{D}$ and $D = D \cap \overline{D}$. If a similar phenomenon occurs between C and some other set, say E, then we may end up with multiple equivalent sets with different names, and represented by different nodes of the topology graph: $C = C \cap \overline{D} = C \cap \overline{E}$.

To identify each of these cases, each of the resulting sets must be checked for vacuity, and uniqueness. No matter the programming language of the algorithm implementation, it is necessary to be implement these two checks.

In Common Lisp there are two possible ways to check for vacuity, *i.e.*, to detect whether a type is empty. (1) Symbolically reduce the type specifier, *e.g.* `(and fixnum (not fixnum))` to a canonical form with is `nil` in case the specifier specifies the `nil` type. (2) Use the `subtypep` function to test whether the type is a subtype of `nil`. To test whether two specifiers specify the same type there are two possible approaches in Common Lisp. (1) Symbolically reduce each expression such as `(or integer number string)` and `(or string fixnum number)` to canonical form, and compare the results with the `equal` function. (2) Use the `subtypep` function twice to test whether each is a subtype of the other.

See Chapter 10 for a description of the performance of this algorithm.

Bibliography

[ACPZ18] Davide Ancona, Giuseppe Castagna, Tommaso Petrucciani, and Elena Zucca. Semantic subtyping for non-strict languages. In 24th International Conference on Types for Proofs and Programs (TYPES 2018), jun 2018.

[Ake78] S. B. Akers. Binary Decision Diagrams. IEEE Trans. Comput., 27(6):509–516, June 1978.

[Als11] Gerold Alsmeyer. Chebyshev's Inequality, pages 239–240. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[Ami16] Nada Amin. Dependent Object Types. page 134, 2016.

[And99] Henrik Reif Andersen. An introduction to binary decision diagrams. Technical report, Course Notes on the WWW, 1999.

[Ano87] Anonymous. The Common Lisp Object Standard (CLOS), October 1987. 1 videocassette (VHS) (53 min.).

[ANO+12] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Valentin Mayer-Eichberger. A New Look at BDDs for Pseudo-Boolean Constraints. J. Artif. Intell. Res., 45:443–480, 2012.

[Ans94] Ansi. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.

[AS96] Harold Abelson and Gerald J. Sussman. Structure and Interpretation of Computer Programs. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[ATB94] Adnan Aziz, Serdar Taşiran, and Robert K. Brayton. BDD Variable Ordering for Interacting Finite State Machines. In Proceedings of the 31st Annual Design Automation Conference, DAC '94, pages 283–288, New York, NY, USA, 1994. ACM.

[Aug85] Lennart Augustsson. Compiling Pattern Matching. In Proc. Of a Conference on Functional Programming Languages and Computer Architecture, pages 368–381, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[Bak92] Henry G. Baker. A Decision Procedure for Common Lisp's SUBTYPEP Predicate. Lisp and Symbolic Computation, 5(3):157–190, 1992.

[Bar87] Jeff Barnett. Types in CL. Computer History Museum Sofware Preservation Group, December 1987.

[Bar90] T.J. Barnes. SKILL: a CAD system extension language. In Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE, pages 266–271, Jun 1990.

[Bar09] Jeff Barnett. The CRISP Programming Language System, An Historical Overview. Computer History Museum Sofware Preservation Group, September 2009.

[Bar10] Jeff Barnett. Notes on SDC CRISP for IBM 370 Computers. Computer History Museum Sofware Preservation Group, June 2010.

[Bar15] Daniel Barlow. ASDF User Manual for Version 3.1.6, 2015.

[Bar18] Jeff Barnett. searching for Jeff Barnett. conversation on comp.lang.lisp, August 2018.

[BBDC+18] Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. Java & Lambda: a Featherweight Story. Logical Methods in Computer Science, 2018. to appear.

[BC16] David Bergman and Andre A. Cire. Theoretical Insights and Algorithmic Tools for Decision Diagram-Based Optimization. Constraints, 21(4):533, 2016.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 1020 States and Beyond. Inf. Comput., 98(2):142–170, June 1992.

[BDG+88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System specification. ACM SIGPLAN Notices, 23(SI):1–142, 1988.

[BF] Robert Brown and coise-René Rideau Fran˙ Google Common Lisp Style Guide, Revision 1.28. accessed 14 October 2018, 12h36 +0200.

[BG84] Rodney A. Brooks and Richard P. Gabriel. A Critique of Common Lisp. In LISP and Functional Programming, pages 1–8, 1984.

[BKK+86] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, Stefik, M., and F. Zdybel. Common Loops, merging Lisp and object-oriented programming. j-SIGPLAN, 21(11):17–29, November 1986.

[Bla15] Jim Blandy. The Rust Programming Language: Fast, Safe, and Beautiful. O'Reilly Media, Inc., 2015.

[Bou17] Pascal J. Bourguignon. I'm annoyed by the specification for satisfies. Thread on comp.lang.lisp, January 2017.

[Bou18] Pascal J. Bourguignon. seeding random number generation different every time i run the program, comp.lang.lisp, 2018.

[BP74] Jeff Barnett and D. L. Pintar. CRISP: A Programming language and System (draft), December 1974.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a bdd package. In 27th ACM/IEEE Design Automation Conference, pages 40–45, Jun 1990.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers, 35:677–691, August 1986.

[Bry92] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-decision Diagrams. ACM Comput. Surv., 24(3):293–318, September 1992.

[Bry18] Randal E. Bryant. Chain Reduction for Binary and Zero-Suppressed Decision Diagrams. In Dirk Beyer and Marieke Huisman, editors, Tools and Algorithms for the Construction and Analysis of Systems, pages 81–98, Cham, 2018. Springer International Publishing.

[Brz64] Janusz A. Brzozowski. Derivatives of Regular Expressions. J. ACM, 11(4):481–494, October 1964.

[Cam99] Robert D. Cameron. Perl Style Regular Expressions in Prolog, CMPT 384 Lecture Notes, 1999.

[Cas16] Giuseppe Castagna. Covariance and Contravariance: a fresh look at an old issue. Technical report, CNRS, 2016.

[CB14] Paul Chiusano and Rnar Bjarnason. Functional Programming in Scala. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2014.

[CBM90] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of Synchronous Sequential Machines Based on Symbolic Execution. In Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems, pages 365–373, London, UK, UK, 1990. Springer-Verlag.

[CD80] Mario Coppo and Mariangiola Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. Notre Dame Journal of Formal Logic, 21(4):685–693, 1980.

[CD10] Pascal Costanza and Theo D'Hondt. Embedding Hygiene-Compatible Macros in an Unhygienic Macro System. Journal of Universal Computer Science, 16(2):271–295, jan 2010.

[CDCV] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional Characters of Solvable Terms. Mathematical Logic Quarterly, 27(2-6):45–58.

[CF05] Giuseppe Castagna and Alain Frisch. A Gentle Introduction to Semantic Subtyping. In Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP '05, pages 198–199, New York, NY, USA, 2005. ACM.

[CH05] Pascal Costanza and Robert Hirschfeld. Language Constructs for Context-oriented Programming: An Overview of ContextL. In Proceedings of the 2005 Symposium on Dynamic Languages, DLS '05, pages 1–10, New York, NY, USA, 2005. ACM.

[Che17] Tongfei Chen. Typesafe Abstractions for Tensor Operations. CoRR, abs/1710.06892, 2017.

[Chr09] Juliusz Chroboczek. CL-Yacc, a LALR(1) parser generator for Common Lisp, 2009.

[Chu41] Alonzo Church. The Calculi of Lambda-Conversion. Princeton University Press, Princeton, New Jork, 1941.

[CL17] G. Castagna and V. Lanvin. Gradual Typing with Union and Intersection Types. Proc. ACM Program. Lang., (1, ICFP '17, Article 41), sep 2017.

[CM85] Eugene Charniak and Drew McDermott. Introduction to Artificial Intelligence. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985.

[CMZ+97] E. M. Clarke, K. L. Mcmillan, X. Zhao, M. Fujita, and J. Yang. Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping. Form. Methods Syst. Des., 10(2-3):137–148, April 1997.

[Col13] Maximilien Colange. Symmetry Reduction and Symbolic Data Structures for Model Checking of Distributed Systems. Thèse de doctorat, Laboratoire de l'Informatique de Paris VI, Université Pierre-et-Marie-Curie, France, December 2013.

[com15] Declaring the elements of a list, discussion on comp.lang.lisp, January 2015.

[Cos] Pascal Costanza. Closer project.

[CR91] William Clinger and Jonathan Rees. Macros That Work. In Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '91, pages 155–162, New York, NY, USA, 1991. ACM.

[Cuk17] Ivan Cukic. Functional Programming in C++. Manning Publications, 2017.

[DL15] Alexandre Duret-Lutz. Conversations concerning segmentation of sets, November 2015.

[Dom] Public Domain. Alexandria implementation of destructuring-case.

[dot18] Dotty Documentation, 0.10.0-bin-SNAPSHOT, August 2018.

[DS88] J.D. DePree and C. Swartz. Introduction to real analysis. Wiley, 1988.

[Dun12] Joshua Dunfield. Elaborating Intersection and Union Types. SIGPLAN Not., 47(9):17–28, September 2012.

[FCB08a] A. Frisch, G. Castagna, and V. Benzaken. Semantic Subtyping: dealing set-theoretically with function, union, intersection, and negation types. Journal of the ACM, 55(4):1–64, 2008. Extends and supersedes LICS '02 and ICALP/PPDP '05 articles.

[FCB08b] Alain Frischn, Giuseppe Castagna, and Véronique Benzaken. Semantic Subtyping: Dealing Set-theoretically with Function, Union, Intersection, and Negation Types. J. ACM, 55(4):19:1–19:64, September 2008.

[FTV16] Dror Fried, Lucas M. Tabajara, and Moshe Y. Vardi. BDD-Based Boolean Functional Synthesis. In Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II, pages 402–421, 2016.

[Fun13] Nobuhiko Funato. Public domain implementation of destructuring-bind, 2013. accessed 14 October 2018, 12h36 +0200.

[Gab90] Richard P. Gabriel. Common Lisp Email, August 1990.

[Gal87] Nick Gall. Re: Types in CL. Computer History Museum Sofware Preservation Group, December 1987.

[GF64] Bernard A. Galler and Michael J. Fisher. An improved equivalence algorithm. Commununication of the ACM, 7(5):301–303, may 1964.

[Gos08] William Sealy Gosset. The Probable Error of a Mean. Biometrika, 6(1):1–25, March 1908. Originally published under the pseudonym "Student".

[GPS98] Clemens Gröpl, Hans Jürgen Prömel, and Anand Srivastav. Size and structure of random ordered binary decision diagrams. In STACS 98, pages 238–248. Springer Berlin Heidelberg, 1998.

[GPS01] Clemens Gröpl, Hans Jürgen Prömel, and Anand Srivastav. On the evolution of the worst-case OBDD size. Inf. Process. Lett., 77(1):1–7, 2001.

[Gra96] Paul Graham. ANSI Common Lisp. Prentice Hall Press, Upper Saddle River, NJ, USA, 1996.

[Haz] Philip Hazel. PCRE - Perl Compatible Regular Expressions.

[hd] Dan Doel (https://cs.stackexchange.com/users/93254/dan doel). how to deduce a function subtype rule from a given function type definition. Computer Science Stack Exchange. URL:https://cs.stackexchange.com/q/97342 (version: 2018-09-16).

[HM94] Mark A. Heap and M. R. Mercer. Least Upper Bounds on OBDD Sizes. 43:764–767, June 1994.

[HMR+15] Magne Haveraaen, Karla Morris, Damian Rouson, Hari Radhakrishnan, and Clayton Carson. High-Performance Design Patterns for Modern Fortran. Scientific Programming, page 14, 2015.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Hos00] Haruo Hosoya. Regular Expression Types for XML. PhD thesis, The University of Tokyo, December 2000.

[Hot30] Harold Hotelling. British Statistics and Statisticians Today. 25(170):186–190, June 1930.

[Hoy08] Doug Hoyte. Let Over Lambda. Lulu.com, 2008.

[HP01] Haruo Hosoya and Benjamin Pierce. Regular Expression Pattern Matching for XML. SIGPLAN Not., 36(3):67–80, January 2001.

[Hro02] Juraj Hromkovič. Descriptional Complexity of Finite Automata: Concepts and Open Problems. J. Autom. Lang. Comb., 7(4):519–531, September 2002.

[Hug89] J. Hughes. Why Functional Programming Matters. Comput. J., 32(2):98–107, April 1989.

[Hut99] Graham Hutton. A Tutorial on the Universality and Expressiveness of Fold. J. Funct. Program., 9(4):355–372, July 1999.

[HVP05] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular Expression Types for XML. ACM Trans. Program. Lang. Syst., 27(1):46–90, January 2005.

[iM96] Shin ichi Minato. Springer US, 1996.

[Jos12] Nicolai M. Josuttis. The C++ Standard Library: A Tutorial and Reference. Addison-Wesley Professional, 2nd edition, 2012.

[Kar68] Marvin Karson. Handbook of Methods of Applied Statistics. Volume I: Techniques of Computation Descriptive Methods, and Statistical Inference. Volume II: Planning of Surveys and Experiments. I. M. Chakravarti, R. G. Laha, and J. Roy, New York, John Wiley; 1967, $9.00. Journal of the American Statistical Association, 63(323):1047–1049, 1968.

[Kat15] Douglas Katzman. Thread on SBCL devel-list sbcl-devel@lists.sourceforge.net, December 2015.

[Kay69] Alan C. Kay. The Reactive Engine. PhD thesis, University of Utah, 1969.

[KdRB91] Gregor J. Kiczales, Jim des Rivières, and Daniel G. Bobrow. The Art of the Metaobject Protocol. MIT Press, Cambridge, MA, 1991.

[Kee89] Sonja E. Keene. Object-Oriented Programming in Common Lisp: a Programmer's Guide to CLOS. Addison-Wesley, 1989.

[KK04] Ilias S. Kotsireas and Kostas Karamanos. Exact computation of the Bifurcation point B_4 of the logistic map and the Bailey-Broadhurst conjectures. I. J. Bifurcation and Chaos, 14(7):2417–2423, 2004.

[KKWZ15] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. Learning Spark: Lightning-Fast Big Data Analytics. O'Reilly Media, Inc., 1st edition, 2015.

[Knu09] Donald E. Knuth. The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams. Addison-Wesley Professional, 12th edition, 2009.

[KR88] Brian W. Kernighan and Dennis Ritchie. The C Programming Language, Second Edition. Prentice-Hall, 1988.

[LGN10] Roland Levillain, Thierry Géraud, and Laurent Najman. Why and How to Design a Generic and Efficient Image Processing Framework: The Case of the Milena Library. In Proceedings of the IEEE International Conference on Image Processing (ICIP), pages 1941–1944, Hong Kong, September 2010.

[LGN12] Roland Levillain, Thierry Géraud, and Laurent Najman. Writing Reusable Digital Topology Algorithms in a Generic Image Processing Framework. In Ullrich Köthe, Annick Montanvert, and Pierre Soille, editors, WADGMM 2010, volume 7346 of Lecture Notes in Computer Science, pages 140–153. Springer-Verlag Berlin Heidelberg, 2012.

[LPR03] Michael Langberg, Amir Pnueli, and Yoav Rodeh. The ROBDD Size of Simple CNF Formulas. In Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings, pages 363–377, 2003.

[LS92] Y.-T. Lai and S. Sastry. Edge-valued Binary Decision Diagrams for Multi-level Hierarchical Verification. In Proceedings of the 29th ACM/IEEE Design Automation Conference, DAC '92, pages 608–613, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[LS10] Sergei A. Lozhkin and Alexander E. Shiganov. High Accuracy Asymptotic Bounds on the BDD Size and Weight of the Hardest Functions. Fundam. Inform., 104(3):239–253, 2010.

[LSCCDV12] Laurent Senta, Christopher Chedeau, and Didier Verna. Generic Image Processing with Climb. In European Lisp Symposium, Zadar, Croatia, May 2012.

[Mac92] Robert Maclachlan. Python compiler for CMU Common Lisp. In ACM Sigplan Lisp Pointers, pages 235–246, 01 1992.

[Mac03] Robert Maclachlan. Design of CMU Common Lisp. unpublished, 01 2003.

[Mar98] Lieven Marchand. Singleton pattern in CLOS. Thread on comp.lang.lisp, June 1998.

[Mar08] Luc Maranget. Compiling Pattern Matching to Good Decision Trees. In Proceedings of the 2008 ACM SIGPLAN Workshop on ML, ML '08, pages 35–46, New York, NY, USA, 2008. ACM.

[Mar15] Barry Margolin. declaring the elements of a list. Thread on comp.lang.lisp, December 2015.

[McI60] M. Douglas McIlroy. Macro instruction extensions of compiler languages. Commun. ACM, 3:214–220, April 1960.

[MD03] D. M. Miller and G. W. Dueck. On the size of multiple-valued decision diagrams. In 33rd International Symposium on Multiple-Valued Logic, 2003. Proceedings., pages 235–240, May 2003.

[Min93] Shin-ichi Minato. Zero-suppressed BDDs for Set Manipulation in Combinatorial Problems. In Proceedings of the 30th International Design Automation Conference, DAC '93, pages 272–277, New York, NY, USA, 1993. ACM.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. ACM Trans. Model. Comput. Simul., 8(1):3–30, January 1998.

[MPS84] David MacQueen, Gordon Plotkin, and Ravi Sethi. An Ideal Model for Recursive Polymorphic Types. In Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '84, pages 165–174, New York, NY, USA, 1984. ACM.

[Nat10] Mary Natrella. NIST/SEMATECH e-Handbook of Statistical Methods. NIST/SEMATECH, July 2010.

[NDV16] Jim Newton, Akim Demaille, and Didier Verna. Type-Checking of Heterogeneous Sequences in Common Lisp. In European Lisp Symposium, Kraków, Poland, May 2016.

[New15] William H. Newman. Steel Bank Common Lisp User Manual, 2015.

[New17] Jim Newton. Analysis of Algorithms Calculating the Maximal Disjoint Decomposition of a Set. Technical report, EPITA/LRDE, 2017.

[New18] Jim Newton. What does "argument types are not restrictive" mean? Thread on comp.lang.lisp, August 2018.

[Nor92] Peter Norvig. Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[NP93] Peter Norvig and Kent Pitman. Tutorial on Good Lisp Programming Style. aug 1993. Lisp User and Vendors Conference.

[NV18a] Jim Newton and Didier Verna. A Theoretical and Numerical Analysis of the Worst-Case Size of Reduced Ordered Binary Decision Diagrams. Transactions on Computational Logic, ACM, 2018.

[NV18b] Jim Newton and Didier Verna. Recognizing hetergeneous sequences by rational type expression. In Proceedings of the Meta'18: Workshop on Meta-Programming Techniques and Reflection, Boston, MA USA, November 2018.

[NV18c] Jim Newton and Didier Verna. Strategies for typecase optimization. In European Lisp Symposium, Marbella, Spain, April 2018.

[NVC17] Jim Newton, Didier Verna, and Maximilien Colange. Programmatic Manipulation of Common Lisp Type Specifiers. In European Lisp Symposium, Brussels, Belgium, April 2017.

[Oka98] Chris Okasaki. Purely Functional Data Structures. Cambridge University Press, New York, NY, USA, 1998.

[ORT09a] Scott Owens, John Reppy, and Aaron Turon. Regular-expression Derivatives Re-examined. J. Funct. Program., 19(2):173–190, March 2009.

[ORT09b] Scott Owens, John Reppy, and Aaron Turon. Regular-expression Derivatives Re-examined. J. Funct. Program., 19(2):173–190, March 2009.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. Programming in Scala: A Comprehensive Step-by-step Guide. Artima Incorporation, USA, 1st edition, 2008.

[Pae93] Andreas Paepcke. User-Level Language Crafting – Introducing the CLOS metaobject protocol. In Andreas Paepcke, editor, Object-Oriented Programming: The CLOS Perspective, chapter 3, pages 65–99. MIT Press, 1993. Downloadable version at url.

[PBM10] Md. Mostofa Ali Patwary, Jean R. S. Blair, and Fredrik Manne. Experiments on union-find algorithms for the disjoint-set data structure. In Paola Festa, editor, Proceedings of 9th International Symposium on Experimental Algorithms (SEA'10), volume 6049 of Lecture Notes in Computer Science, pages 411–423. Springer, 2010.

[Pea17] David J. Pearce. Rewriting for sound and complete union, intersection and negation types. In Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, Vancouver, BC, Canada, October 23-24, 2017, pages 117–130, 2017.

[Pie02] Benjamin C. Pierce. Types and Programming Languages. The MIT Press, 1st edition, 2002.

[Pin15] Jean-Éric Pin. Mathematical Foundations of Automata Theory. 2015.

[Pit03] Kent M. Pitman. Using closures with SATISFIES, comp.lang.lisp, 2003.

[Pot80] G. Pottinger. A type assignment for the strongly normalizable lambda-terms. In J. Hindley and J. Seldin, editors, To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pages 561–577. Academic Press, 1980.

[RA16] Tiark Rompf and Nada Amin. Type Soundness for Dependent Object Types (DOT). SIGPLAN Not., 51(10):624–641, October 2016.

[Rey96] John C. Reynolds. Design of the Programming Language Forsythe. Technical report, 1996.

[Rho08] Christophe Rhodes. SBCL: A Sanely-Bootstrappable Common Lisp. In Robert Hirschfeld and Kim Rose, editors, Self-Sustaining Systems, pages 74–86, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Rho09] Christophe Rhodes. User-extensible Sequences in Common Lisp. In Proceedings of the 2007 International Lisp Conference, ILC '07, pages 13:1–13:14, New York, NY, USA, 2009. ACM.

[Rie] Chris Riesbeck. Lisp Unit. https://www.cs.northwestern.edu/academics/courses/325/readings/lisp-unit.html.

[RL17] Marianna Rapoport and Ondřej Lhoták. Mutable Wadlerfest DOT. In Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs, FTFJP'17, pages 7:1–7:6, New York, NY, USA, 2017. ACM.

[Sab11] Miles Sabin. Unboxed union types in Scala via the Curry-Howard isomorphism, June 2011.

[Saj17] Yogesh Sajanikar. Haskell Cookbook: Build functional applications using Monads, Applicatives, and Functors. Packt Publishing Ltd, 2017.

[SCG13] Alex Shinn, John Cowan, and Arthur A. Gleckler. Revised 7 report on the algorithmic language Scheme. Technical report, 2013.

[SG93] Guy L. Steele, Jr. and Richard P. Gabriel. The Evolution of Lisp. In The Second ACM SIGPLAN Conference on History of Programming Languages, HOPL-II, pages 231–270, New York, NY, USA, 1993. ACM.

[SGC15] Don Syme, Adam Granicz, and Antonio Cisternino. Expert F# 4.0. Apress, Berkely, CA, USA, 4th edition, 2015.

[Sha49] C. E. Shannon. The synthesis of two-terminal switching circuits. The Bell System Technical Journal, 28(1):59–98, Jan 1949.

[SIHB97] Tsutomu Sasao, Robert J. Barton III, David S. Herscovici, and Jon T. Butler. Average and Worst Case Number of Nodes in Decision Diagrams of Symmetric Multiple-Valued Functions. IEEE Transactions on Computers, 46:491–494, 1997.

[Sla98] S. Slade. Object-oriented Common LISP. Prentice Hall PTR, 1998.

[Som] Fabio Somenzi. CUDD: BDD package, University of Colorado, Boulder.

[Sri02] A. Srinivasan. Algorithms for discrete function manipulation. In Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on, 2002.

[ST98] Karsten Strehl and Lothar Thiele. Symbolic Model Checking of Process Networks Using Interval Diagram Techniques. In Proceedings of the 1998 IEEE/ACM International Conference on Computer-aided Design, ICCAD '98, pages 686–692, New York, NY, USA, 1998. ACM.

[Ste84] Guy L. Steele, Jr. Common LISP: The Language (1st Ed.). Digital Equipment Corporation, 1984.

[Ste90] Guy L. Steele, Jr. Common LISP: The Language (2nd Ed.). Digital Press, Newton, MA, USA, 1990.

[Str81] Karl R. Stromberg. An Introduction to Classical Real Analysis. Wadsworth International, Belmont, CA, 1981.

[Str13] Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley Professional, 4th edition, 2013.

[Swa17] M. Swaine. Functional Programming: a PragPub Anthology: Exploring Clojure, Elixir, Haskell, Scala, and Swift. Pragmatic programmers. Pragmatic Bookshelf, 2017.

[THFF+17] Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. Migratory Typing: Ten Years Later. In SNAPL, pages 17:1–17:17, 2017.

[Val18] Leo Valais. SUBTYPEP: An Implementation of Baker's Algorithm. Technical report, EPITA/L-RDE, July 2018.

[vR88] Walter van Roggen. Issue FUNCTION-TYPE-ARGUMENT-TYPE-SEMANTICS Writeup, September 1988.

[Wam11] Dean Wampler. Functional Programming for Java Developers: Tools for Better Concurrency, Abstraction, and Agility. O'Reilly Media, Inc., 2011.

[Weg87] Ingo Wegener. The Complexity of Boolean Functions. John Wiley & Sons, Inc., New York, NY, USA, 1987.

[Wei15] Edmund Weitz. Common Lisp Recipes: A Problem-solution Approach. Apress, 2015.

[XAMM93] L. Xu, A. E. A. Almaini, J. F. Miller, and L. McKenzie. Reed-Muller universal logic module networks. IEE Proceedings E - Computers and Digital Techniques, 140(2):105–108, March 1993.

[Xin04] Guangming Xing. Minimized Thompson NFA. Int. J. Comput. Math., 81(9):1097–1106, 2004.

[YD14] Francois Yvon and Akim Demaille. Théorie des Langages Rationnels. EPITA LRDE, 2014. Lecture notes.

[Zab08] S. L Zabell. On Student's 1908 Article "The Probable Error of a Mean". Journal of the American Statistical Association, 103(481):1–7, 2008.