Aggressive Aggregation

(Domain-Specific) Program Optimisation with Algebraic Decision Diagrams

Dissertation
to obtain the Degree

Doctor of Philosophy

from the University of Limerick
at the Department of Computer Science and Information Systems
Limerick, Ireland

Dissertation
zur Erlangung des Grades

Doktor der Ingenieurwissenschaften

der Technische Universität Dortmund
an der Fakultät für Informatik
Dortmund, Germany

Frederik Jakob Gossen

2021

Supervisor at the University of Limerick (lead institution)
Prof. Tiziana Margaria

Supervisor at the TU Dortmund University
Prof. Bernhard Steffen

Acknowledgement

Like with anything that's a labour of dedication and tremendous effort, the success of this PhD was a result of the unwavering support of many people.

First and foremost, I would like to equally thank my supervisors Prof. Tiziana Margaria at the University of Limerick and Prof. Bernhard Steffen at the TU Dortmund University. Throughout my research work, they have consistently inspired and supported me, to a degree that is exceptional for a PhD supervision. Without their insight and motivation, this thesis would have not been completed.

I would also like to extend gratitude to Prof. Falk Howar and Prof. Mike Hinchey for the many fruitful discussions concerning this work - I have benefitted immensely from their feedback and invaluable advice provided throughout this journey.

As a member of Lero - the Irish Software Research Centre, I had the gracious opportunity to connect with and learn alongside esteemed researchers. For this, I would like to thank Dr. Anna-Lena Lamprecht, Prof. Wylliams Santos, Dr. Yehia Elrakaiby, and Dr. George Grispos for their listening ears and open minds that were vital sources of inspiration throughout the early phase of my journey.

Being a part of both universities, the University of Limerick and the TU Dortmund University, provided me with the chance to flourish in academically-intensive environments. This dynamic collaboration in aligning the procedures of the two universities was a success due to the work of my supervisors and also Prof. Jeff Punch, Michael Frain, and Ulrike Herberholz. I wholeheartedly thank you for your efforts. This joint degree is a result of your will and dedication.

On this x year journey, I was afforded the opportunity to connect with researchers and PhD candidates across borders, many of whom have become friends over time. Thus, in addition to those already mentioned, I would also like to thank Marc Jasper, Alnis Murtovi, Tim Tegeler, Malte Mues, and Dr. Stefan Naujokat.

Finally, to my family members and to my partner Shaye who supported me quietly in the background and provided strength in the moments where times were hard, my sincerest gratitude.

This work came to fruition as a result of all of our efforts. Thank you.

Abstract

Among the first steps in a compilation pipeline is the construction of an Intermediate Representation (IR), an in-memory representation of the input program. Any attempt to program optimisation, both in terms of size and running time, has to operate on this structure. There may be one or multiple such IRs, however, most compilers use some form of a Control Flow Graph (CFG) internally. This representation clearly aims at general-purpose programming languages, for which it is well suited and allows for many classical program optimisations. On the other hand, a growing structural difference between the input program and the chosen IR can lose or obfuscate information that can be crucial for effective optimisation. With today's rise of a multitude of different programming languages, Domain-Specific Languages (DSLs), and computing platforms, the classical machine-oriented IR is reaching its limits and a broader variety of IRs is needed. This realisation yielded, e.g., Multi-Level Intermediate Representation (MLIR), a compiler framework that facilitates the creation of a wide range of IRs and encourages their reuse among different programming languages and the corresponding compilers.

In this modern spirit, this dissertation explores the potential of Algebraic Decision Diagrams (ADDs) as an IR for (domain-specific) program optimisation. The data structure remains the state of the art for Boolean function representation for more than thirty years and is well-known for its optimality in size and depth, i.e. running time. As such, it is ideally suited to represent the corresponding classes of programs in the role of an IR. We will discuss its application in a variety of different program domains, ranging from DSLs to machine-learned programs and even to general-purpose programming languages.

Two representatives for DSLs, a graphical and a textual one, prove the adequacy of ADDs for the program optimisation of modelled decision services. The resulting DSLs facilitate experimentation with ADDs and provide valuable insight into their potential and limitations: input programs can be aggregated in a radical fashion, at the risk of the occasional exponential growth. With the aggregation of large Random Forests into a single aggregated ADD, we bring this potential to a program domain of practical relevance. The results are impressive: both running time and size of the Random Forest program are reduced by multiple orders of magnitude. It turns out that this ADD-based aggregation can be generalised, even to general-purpose programming languages. The resulting method achieves impressive speedups for a seemingly optimal program: the iterative Fibonacci implementation.

Altogether, ADDs facilitate effective program optimisation where the input programs allow for a natural transformation to the data structure. In these cases, they have proven to be an extremely powerful tool for the optimisation of a program's running time and, in some cases, of its size. The exploration of their potential as an IR has only started and deserves attention in future research.

Attached Publications

Parts of this dissertation were already published in cooperation with other scientists. These publications with comments on my participation are listed below.

I F. Gossen and B. Steffen
Algebraic Aggregation of Random Forests: Towards Explainability and Rapid Evaluation
In: International Journal on Software Tools for Technology Transfer (accepted). 2021.

Cited as [1]$_{\text{AP}}$.

The presented ideas were discussed among all authors. I was the main author of sections 2 through 10 and co-authored all other sections.

II F. Gossen, T. Margaria, and B. Steffen
Towards Explainability in Machine Learning: The Formal Methods Way
In: IT Professional, vol. 22. 2020.

Cited as [2]$_{\text{AP}}$.

The presented ideas were discussed among all authors and build up on the ideas presented in [3]$_{\text{AP}}$.

III F. Gossen and B. Steffen
Large Random Forests: Optimisation for Rapid Evaluation
In: CoRR. 2019.

Cited as [3]$_{\text{AP}}$.

The presented ideas were discussed among all authors. I was the main author of sections 2 through 6 and co-authored all other sections.

IV F. Gossen, M. Jasper, A. Murtovi, and B. Steffen
Aggressive Aggregation: a New Paradigm for Program Optimization
In: CoRR. 2019.

Cited as $[4]_{\mathrm{AP}}$.

The presented ideas were discussed among all authors. I was the main author of sections 3 and 4 and co-authored all other sections.

V F. Gossen, A. Murtovi, P. Zweihoff, and B. Steffen
ADD-Lib: Decision Diagrams in Practice
In: CoRR. 2019.

Cited as $[5]_{\mathrm{AP}}$.

The presented ideas were discussed among all authors. I was the main author of sections 2, 3, and 5 and co-authored all other sections.

VI B. Steffen, F. Gossen, S. Naujokat, and T. Margaria
Language-Driven Engineering: From General-Purpose to Purpose-Specific Languages
In: Computing and Software Science. Lecture Notes in Computer Science, vol. 10000. 2019.

The presented ideas were discussed among all authors. I am the main author of sections 3 and 4 and realised the implementation of all example languages.

Cited as $[6]_{\mathrm{AP}}$.

VII F. Gossen, T. Margaria, A. Murtovi, S. Naujokat, and B. Steffen
DSLs for Decision Services: A Tutorial Introduction to Language-Driven Engineering
In: Leveraging Applications of Formal Methods, Verification and Validation. Modeling. ISoLA 2018. Lecture Notes in Computer Science, vol. 11244. 2018.

Cited as $[7]_{\mathrm{AP}}$.

The presented ideas originate in $[6]_{\mathrm{AP}}$ and were discussed among all authors. I am the main author of sections 2 and 3.

VIII F. Gossen and T. Margaria
Generating Optimal Decision Functions from Rule Specifications
In: Electronic Communications of the EASST, vol. 74. 2017.

Cited as $[8]_{\mathrm{AP}}$.

The presented ideas originate [9] and were discussed among both authors. I am the main author of sections 2 through 7.

Abbreviations

ADD Algebraic Decision Diagram. iii, iv, , 3–5, 12–14, 17–19, 21–32, 34–36, 38–43, 50–56, 60, 61, 63–67, 69–71, 73, 74

BDD Binary Decision Diagram. , 8–15, 39, 40, 51, 52, 61

BDP Binary Decision Program. , 7, 8, 65, 67

BNF Backus-Naur Form. , 42

BNN Binary Neural Network. , 5, 70, 71

CDD Chain-Reduced Decision Diagram. , 15

CFG Control Flow Graph. iii, , 1, 3

CPU Central Processing Unit. , 2, 74

DAG Directed Acyclic Graph. , 8, 9, 21, 52, 70

DNN Deep Neural Network. , 2, 5, 39, 70

DSL Domain-Specific Language. iii, , 2–5, 17–19, 21–23, 26, 27, 41, 69, 70, 73

ED Expression DAG. , 41–43, 54, 56, 60, 71, 72

GPU Graphics Processing Unit. , 2, 70, 72, 74

IR Intermediate Representation. iii, iv, , 1–5, 17, 18, 21, 28, 38, 41, 42, 69, 71, 73, 74

LDE Language-Driven Engineering. , 18

LLVM Low Level Virtual Machine.

LOP Language-Oriented Programming. , 18

mIDE Mindset-Supporting Integrated Development Environment. , 18, 19

ML Machine Learning. , 2, 39

MLIR Multi-Level Intermediate Representation. iii, , 2

MTBDD Multi-Terminal Binary Decision Diagram. , 12

MVDD Multi-Value Decision Diagram. , 14

OBDD Ordered Binary Decision Diagram. , 8–10, 12

PLC Programmable Logic Controller. , 71

ROBDD Reduced Ordered Binary Decision Diagram. , 9–12

SOS Structural Operational Semantics. , 43, 45–49

SSA Static Single Assignment. , 1, 41

ZDD Zero-Suppressed Decision Diagram. , 14, 15, 61

Contents

1 Introduction **1**
1.1 Scientific Contributions 3
1.2 Overview 5

2 Landscape of Decision Diagrams **7**
2.1 The History of Decision Diagrams 7
2.2 Binary Decision Diagrams 9
2.3 Algebraic Decision Diagrams 12
2.4 Other Variants of Decision Diagrams 14

3 (Domain-Specific) Program Optimisation with Algebraic Decision Diagrams **17**
3.1 Domain-Specific Languages 18
3.2 A Machine-Learned Program: The Random Forest 26
3.3 Towards Explainability 39
3.4 General-Purpose Programming Languages 41

4 ADD-Lib: A Highly Flexible Framework for Decision Diagrams and Code Generation **61**
4.1 Algebra as a Service: Algebraic Decision Diagrams in the ADD-Lib 63
4.2 Decision Diagram Visualisation 65
4.3 Code Generation: From Decision Diagrams to Executable Code 65

5 Future Work **69**
5.1 Domain-Specific Languages 69
5.2 Machine-Learned Models 70
5.3 General-Purpose Programming Languages 71

6 Conclusion **73**

1 Introduction

Among the first steps in a compilation pipeline is the construction of an Intermediate Representation (IR), an in-memory model of the input program. On this basis, a compiler analyses the input program and infers static knowledge that it then uses to optimise the program, both in terms of expected execution time and also size. There may be one or multiple IRs subsequent to each other in the pipeline, however, in most modern compilers such as LLVM-based compilers [10][11], these IRs have one thing in common [12][13, 14]: they are variants of Control Flow Graphs (CFGs), often in Static Single Assignment (SSA) form [15, 16, 17]. These graphs aim at general-purpose programming languages for which they are well suited. They represent the program without loss of information and, at the same time, allow for classical program optimisations [18, 19, 20, 21, 22, 23].

Any attempt at optimisation will have to target one of the IRs and, as a consequence, can only exploit properties of the input program that remain at this stage. For programs written in a general-purpose language that is structurally similar to the chosen IR, most of the original structure retains and consequently, the various optimisations can fully exploit the program structure. With the growing structural differences between input language and IR, however, properties can already be lost or obfuscated early in the compilation pipeline.

Consider, for example, matrix transposition: A duplicate matrix transposition is clearly redundant as it has no side-effects and the result will be the same as the original matrix. This redundancy is easy to detect when the matrix transposition is explicit, however, when compiled down to a loop nest, it is difficult for a compiler to recognise and thus would require expensive analysis.

As a consequence, program optimisations are unable to detect patterns that can be crucial for effective running time or even size improvements. A solution to the problem can be multiple subsequent IRs that each allow for their respective optimisations techniques. The key to success is, as so often, the *right* representation. Many modern programming languages use LLVM as their compiler backend, however, they often implement their own language-specific IR upfront that allows them to tackle very specific properties before relying on more general implementations in LLVM. When done right, this progressive lowering [24] maintains program information as long as required and lowers the program in multiple steps until it has become highly performant

```
func @duplicate_transpose(%arg : tensor<?x?xf32>) -> tensor<?x?xf32> {
  %perm = constant dense<[1, 0]> : tensor<2xi32>
  %0 = "tf.Transpose"(%arg, %perm)
      {T = "tfdtype$DT_FLOAT", Tperm = "tfdtype$DT_INT32"}
      : (tensor<?x?xf32>, tensor<2xi32>) -> tensor<?x?xf32>
  %1 = "tf.Transpose"(%0, %perm)
      {T = "tfdtype$DT_FLOAT", Tperm = "tfdtype$DT_INT32"}
      : (tensor<?x?xf32>, tensor<2xi32>) -> tensor<?x?xf32>
  return %1 : tensor<?x?xf32>
}
```

Figure 1.1: Exemplary TensorFlow IR: A redundant duplicate transposition of a matrix.

executable machine code.

With the growing landscape of programming languages and computation platforms, this pattern has become so successful that Multi-Level Intermediate Representation (MLIR) [25] [24] has emerged as a compiler framework aiming at a wide variety of IRs. The framework facilitates the introduction of many different IRs and encourages their reuse among programming languages and the respective compilers. Its prime users today are Machine Learning (ML) compilers, which aim to compile programs that are heavy on linear algebra and, as a consequence, typically data flow-oriented and highly parallelisable. Their final use is often the device-accelerated training of Deep Neural Networks (DNNs). Hence, it comes at no surprise that they also target a wide range of machine architectures, ranging from Central Processing Units (CPUs) to Graphics Processing Units (GPUs) and other accelerators.

A concrete example of this is TensorFlow [26] which, at its core, comprises a compiler that is quite different from those for general-purpose programming languages. Its IR, the TensorFlow graph, is mostly data flow-oriented. The choice of this IR is crucial to TensorFlow's success in optimising a given program as it allows it to detect patterns like the aforementioned duplicate tensor transposition, which could easily be obfuscated in a control flow-oriented IR. Figure 1.1 shows the MLIR-based representation of the redundant transposition. Here, it is quite obvious how to simplify the given program. Imagine, on the other hand, an equivalent loop nest that instead copies the elements of the involved tensors: finding the inherent redundancy is nearly impossible and at the very least, extremely expensive.

While the representation of higher-level operations unlocks new optimisations, this all remains an extremely local approach to optimisation. Compiler frameworks like LLVM and MLIR do not grasp the entire semantics of their input programs. Rather, they apply extremely localised patterns, usually considering only a few neighbouring operations, while missing out on greater optimisation opportunities that require a more holistic understanding. With the lack of specific knowledge about their input programs, this heuristic approach is likely the best outcome and it has proven successful in their respective domains.

In sharp contrast to such local approaches, this dissertation will explore the potential of more aggressive optimisation techniques. We will focus on Domain-Specific Languages (DSLs) and exploit the domain knowledge to aggregate the semantics of entire programs into a single concise data structure: the Algebraic Decision Diagram (ADD) [27, 28]. To this aim, the input program is decomposed into its essential aspects, which are subsequently reaggregated in a semantics-preserving fashion. While the decomposed program fragments differ greatly from domain to domain, the aggregated program will always be some form of an ADD. This decomposition and reaggregation is so holistic that any adjacent parts of the input program will almost certainly be separated in the process.

The effect of this aggressive aggregation is twofold:

- The condensed representation of the program is often much smaller than the input program and allows for a much faster evaluation of its semantics. In this way, we radically optimise the program's running time and size, the two of the main goals of any compiler. These improvements are impressive and can reach several orders of magnitude $[1, 3]_{\mathrm{AP}}$.

- The aggregation is also a concise representation of the program semantics. This reduction to the essence can allow for a much better understanding of the program. A prime example for this are machine-learned models, which can perform well in terms of their purpose but can be hard or impossible to understand at the same time. We applied our radical aggregation to Random Forests, a model that is considered uninterpretable, with impressive results. The aggregation reduces the obfuscated model to a single interpretable ADD. With this, we can solve the three explanation problems for Random Forests: model explanation, class characterisation, and outcome explanation [29].

ADDs are well suited to represent programs that resemble decision functions, in part or as a whole. In this case, the data structure has great advantages over other CFG-based forms of IRs: It is extremely well understood [27, 28] and comes with efficient manipulation algorithms. Its properties are also ideal for program optimisation as they ensure minimal size and depth of the structure, which correspond to the size and the running time of the aggregated program. The algebraic nature of ADDs and their compositionality further lead to elegant transformations of the program, allowing for partial aggregation and even for semantic abstraction.

The idea of aggressive aggregation originated out of a desire for rapid facial classification in a previous research project [30, 31] and evolved into a much more general analysis. We will analyse and discuss its application to a variety of different program domains ranging from DSLs in recommender systems to machine-learned programs and even to a general-purpose programming language.

1.1 Scientific Contributions

My scientific contributions in the course of this dissertation (i) explore the potential of Algebraic Decision Diagrams (ADDs) as an Intermediate Representation (IR) for program optimisation in diverse program domains, (ii) solve three prevailing explanation problems for Random Forests, and (iii) facilitate future work in this area through corresponding open-source implementations of decision diagrams, transformations, and code generation. In particular, this dissertation makes the following contributions:

- **The exploration of ADD-based compilation and optimisation for suitable DSLs.** Tailored to ADDs as their IR, the graphical DSLs for decision diagrams and their compositions $[6, 7]_{\text{AP}}$ constitute an initial case study. These languages allow their users to develop decision services in a model-driven fashion and without any base knowledge of programming languages. At the same time, they are a great tool to get a feeling for the potential and the limitations of ADDs as an IR. With their language design, they exploit the inherent optimisation potential of ADDs and integrate seamlessly with other graphical DSLs as well as with Java applications. The proposed languages are also extremely flexible with regard to the underlying algebraic structures and allow for an entire family of languages in a generative fashion $[7]_{\text{AP}}$. The case studies prove the usefulness of ADDs in an ecosystem of DSLs and their seamless integrability into corresponding languages while, at the same time, maintaining their inherent optimality.

  Complementing the graphical DSLs $[6, 7]_{\text{AP}}$, ADD-based program optimisation was also applied in another case study: the textual rule-based MiAamics DSL. Its language design was previously found to be useful in the context of recommender systems [32, 33] and the re-implementation and evaluation that are part of this dissertation prove, once again, the power of ADDs as an IR for program optimisation.

- **ADD-based aggregation of Random Forests.** The main contribution is a technique for the radical aggregation of extremely large Random Forests into a single semantically equivalent ADD $[3, 1, 2]_{\text{AP}}$. Regarding the Random Forest as an input program, the aim of its optimisation is to reduce both running time and size of the represented function. The impressive reduction in running time and size, both by multiple orders of magnitude, prove the effectiveness of ADDs as an IR for the optimisation in this very program domain.

  The technique can even be combined with the graphical DSLs for the same type of ADDs $[5]_{\text{AP}}$, allowing users to combine the power of machine-learning with expert knowledge. Both Random Forests and ADDs models, can be composed, forming machine-learned and expert-enhanced decision functions.

- **The solution to three explanation problems for Random Forests.** Random Forests are considered an uninterpretable black box model in machine learning [29] that lacks explainability on multiple levels. By aggregating the convoluted model into a single ADD, we are representing its semantics concisely. Like a decision tree, the resulting ADD is interpretable and solves the first of three explanation problems directly: the model explanation $[1, 2]_{\text{AP}}$. Exploiting the algebraic nature of ADDs, we can go further: Through straightforward abstraction it is possible to solve a second explanation problem: the class characterisation, which explains the model relative to one of the possible outcomes. These ADDs also allow for a redundancy-free outcome explanation by aggregation of predicates along paths for any given input sample.

- **A generalisation of ADD-based compilation and optimisation for general-purpose programming languages:** With the ADD-based aggregation being so successful for Random Forests, we generalised the approach to general-purpose programming languages, exemplified by the *while* language $[4]_{\text{AP}}$. The generalised technique is, in principle, able to reproduce the exact same results and is, in this sense, a true generalisation. Here, we use multiple ADDs to represent sequential execution between cut points of the input program, an expression DAG to represent arithmetic expressions, and a contracted cut point graph to trace through the program. This use of three very specific IRs allows us to holistically restructure the input program and yields impressive first results for the seemingly optimal Fibonacci implementation, especially when combined with loop unrolling.

- **The ADD-Lib, a Java library and framework for ADDs and code generation.** All reusable components of the corresponding implementations are bundled in this open-source project. The ADD-Lib emphasises interchangeability of the ADDs' underlying algebraic structure and provides an easy to use, yet efficient, interface for a wide variety of decision diagrams, and for ADDs in particular. With its collection of code generators, the framework facilitates the use of ADDs in the context of program optimisation and, in this way, encourages researchers to explore the potential of ADDs as an IR far beyond the results of this dissertation.

My contributions have already inspired other scientists to work on related research and to further explore the potential of ADDs: Alnis Murtovi and Marc Jasper have worked with me on ADD-based compilation of general-purpose programming languages at the TU Dortmund University and are following up with this line of research. Also at the TU Dortmund University, the student researchers Jan Linden and Jan Feider are exploring the potential of ADDs in the context of Deep Neural Network (DNN) and Binary Neural Network (BNN). Another group of researchers around Prof. Tiziana Margaria at the University of Limerick focus on ADD-based compilation in the context of model-driven engineering, following up on the graphical ADD-based DSLs and the ideas of $[6, 7]_{\text{AP}}$.

1.2 Overview

Chapter 2 reviews decision diagrams in depth, starting at their historic roots. The most popular variants and generalisations of the data structure will be discussed, each with their key characteristics in the context of this dissertation. The main chapter, Chapter 3, presents successful ADD-based program optimisations in three program domains: DSLs, Random Forests, and general-purpose programming languages. Chapter 4 introduces the ADD-Lib, our framework for decision diagrams that was developed and used in the course of this dissertation. We will discuss its key features in the context of ADD-based program optimisation including, in particular, code generation. Chapter 5 points out directions of future research, motivated by the results of this dissertation. We conclude in Chapter 6 with final remarks.

6

2

Landscape of Decision Diagrams

The most widely known form of decision diagrams are, of course, (Ordered) Binary Decision Diagrams [34]. For more than thirty years now, this data structure remains state of the art for the representation of Boolean functions. Every computer science student comes into touch with these diagrams, already within the very first semesters of their course. There are, however, many more variants of decision diagrams, some of which are generalisations, others are adaptations. In this chapter, we will discuss the most important variants in the context of this dissertation and we will point the reader to relevant literature for further detail.

2.1 The History of Decision Diagrams

Although the most popular variant, (Ordered) Binary Decision Diagrams [34] are not the first occurrence of the data structure. They mark, however, an important breakthrough in their development and facilitate their use in various domains [35, 36, 37, 38, 39, 40, 41, 42] as well as the development of many variants [43, 44, 45, 46, 47].

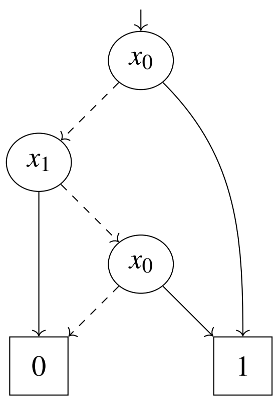
The first form of decision diagrams are, in fact, Binary Decision Programs (BDPs) [48] which were motivated primarily with the faster evaluation of Boolean formulas. Given some number of Boolean input variables $\mathbf{x} = x_0, \ldots, x_{n-1} \in \mathbb{B}^n$, these programs compute an equally Boolean output $y \in \mathbb{B}$. BDPs are essentially a list of labelled if-then-else statements, each evaluating one of the Boolean input variables x_i and jumping accordingly to one of two successor labels until a final decision y is reached.

Figure 2.1a shows the example BDP for the function $x_0 \lor (\neg x_1 \land x_0)$. The individual lines resemble if-then-else statements, e.g. $1 : T x_0; 2, I$ is the first statement and tests the variable x_0. If $x_0 = 1$ the program continues with statement 2, otherwise, the result is given by I. I and θ play a special role and mark the final outcomes true and false respectively. These programs, although textual, are structurally very similar to later graphical presentations of decision diagrams.

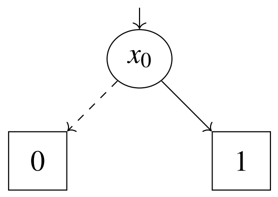
In contrast to Boolean formulas, BDPs are "not algebraic in nature" [48], a property that aggravates their manipulation. Moreover, similarly to *goto* programs, BDPs would be considered hardly readable [49]. From a performance perspective, however, they are easily superior to the original representation, answering the primary question of the paper: "What procedure should

$$1: T \quad x_0; 2, I.$$
$$2: T \quad x_1; 3, \theta.$$
$$3: T \quad x_0; \theta, I.$$

(a) BDP.



(b) Binary Decision Diagram (BDD).



(c) Ordered Binary Decision Diagram (OBDD).

Figure 2.1: Three representations of the Boolean formula $x_0 \vee (\neg x_1 \wedge x_0)$.

one follow so as to arrive at the decision quickly and without having to go through a large amount of computation?" [48].

Based on the early textual form, the first graph-based understanding was derived and the term "Binary Decision Diagram" was coined [50]. BDDs were designed to provide a means for an "'implementation-free' description which could still yield meaningful results about the logical structure" [50]. In other words, the structure seeked to bridge the gap between a functional description (What) and an optimised implementation (How).

BDDs are essentially analogous to BDPs with the key differences that they (i) represent Boolean functions as Directed Acyclic Graphs (DAGs), and they (ii) come with powerful reduction rules. In particular, isomorphic nodes can be merged and those nodes with two equal successors can be removed without altering semantics.

Figure 2.1b shows an example BDD that is fully reduced in this sense. The represented formula is the same as for the BDP (Fig. 2.1a) and the two representations are, in this case, fully analogous. Note, that the representation is by no means optimal, neither in size nor in depth. In fact, input variables can occur even multiple times on a single path leading to obvious redundant computations. The reduction rules, however, constitute a significant improvement over BDPs and are still used today.

Only with the introduction of a fixed variable order, a notion of optimality could finally be guaranteed. These OBDDs mark a breakthrough in the development of decision diagrams and are the state of the art data structure that we use today [34]. For a fixed variable order, OBDDs are a canonical representation of Boolean functions [34]. From this property, their optimality in size and in depth follows trivially. More importantly, the order allows for efficient operations on the data structure, achieving the long-seeked "algebraic nature" [48] for the data structure.

Interestingly, the many examples in earlier BDDs often implicitly conform to a variable order already [50]. It was not until [34] that this property was enforced and used for efficient algorithms and their main result: canonicity.

Figure 2.1c shows an example OBDD for the same example Boolean formula as before. In contrast to the other representations in Figure 2.1, the strict variable order allows for exhaustive simplification of the diagram. The resulting OBDD is not only smaller but also shallower and, as a result, the underlying Boolean formula can be evaluated much faster.

2.2 Binary Decision Diagrams

As the most widely known and relatively simple variant, BDDs will serve as an example to discuss decision diagrams and their algorithms in more detail. Most importantly, we will see how these diagrams can be constructed from Boolean formulas and how Boolean operations can be efficiently realised directly on the data structure. We will formalise BDDs analogous to the formalisation in [34] and discuss, in particular, the functions $unique$, ite, and $apply$.

A BDD is a graph-based representation for Boolean functions of the form $\mathbb{B}^n \to \mathbb{B}$. In fact, every node in the graph is associated with its own function, so when we refer to a BDD, we typically mean a root node in the graph together with its transitive closure.

Definition 1 (Binary Decision Diagram) *A BDD over the input variables $\mathbf{x} = x_0, \ldots, x_{n-1}$ is a DAG with nodes V of two kinds: (i) Internal nodes $f \in V$ are associated with a variable $var(f) \in \mathbf{x}$ and have exactly two successor nodes, $then(f), else(f) \in V$*[1]*. (ii) Terminal nodes $g \in V$, on the other hand, are associated with a Boolean $value(g) \in \mathbb{B}$ and have no successor nodes.*

The semantics of these graphs follow naturally. Starting at the root node, we trace down the diagram until a terminal node is reached. At every internal node f, the associated variable $var(f)$ determines the successor and once a terminal g is reached, the associated Boolean $value(g)$ is also the final evaluation result.

Definition 2 (Binary Decision Diagram Semantics) *For a given assignment of the BDD's variables $\sigma : \mathbf{x} \to \mathbb{B}$, the semantics of its nodes f are defined as*

$$[\![f]\!]_\sigma := \begin{cases} [\![then(f)]\!]_\sigma & \text{if } f \text{ is an internal node and } \sigma(var(f)) = 1, \\ [\![else(f)]\!]_\sigma & \text{if } f \text{ is an internal node and } \sigma(var(f)) = 0, \\ value(f) & \text{if } f \text{ is a terminal node.} \end{cases}$$

Already with this definition, we have a graphical representation of Boolean formulas that even allows for their composition with the usual logic operations. In this form, however, the data structure lacks the potential for simplification and for efficient evaluation. With the introduction of a fixed variable order and effective reduction rules, Reduced Ordered Binary Decision Diagrams (ROBDDs) are finally able to achieve canonicity for Boolean functions.

Let us first consider their fixed variable order. On every path in an OBDD the variables that are associated with its internal nodes appear in a predefined order. This ensures, in particular, that a variable may never appear more than once on a path and thus prevents some redundancies, already on its own.

[1] In literature successor nodes are also referred to as $high$ and low respectively.

Definition 3 (Ordered Binary Decision Diagram) *For a fixed variable order $<$, a BDD is ordered if and only if for every internal node f the following holds: If $then(f)$ is also an internal node then $var(f) < var(then(f))$ and analogously if $else(f)$ is an internal node then $var(f) < var(else(f))$.*

In addition, we want to simplify the OBDD wherever possible. The aim is to maintain, at all times, a representation that is fully reduced.

Definition 4 (Reduced Binary Decision Diagram) *A BDD is reduced if and only if (i) for every internal node f its successors differ, $then(f) \neq else(f)$, and (ii) there are no two isomorphic subparagraphs.*

These two properties ensure that none of the nodes in a ROBDD is redundant. Transforming an OBDD to its equivalent ROBDD is straightforward [34]. Potential duplicate nodes can simply be merged into one and those nodes whose successors are equal can be eliminated entirely.

With their fixed variable order and being fully reduced, ROBDDs are a canonical and minimal representation of Boolean functions — the main theorem of BDDs [34]:

Theorem 1 (Reduced Ordered Binary Decision Diagrams are Canonical) *Let $<$ be a fixed variable order. For every Boolean function f, there is a unique (up to isomorphism) ROBDD. Every other OBDD contains more nodes.*

With these properties ROBDDs are the desirable graph representation for Boolean functions. Moreover, they are algebraic in nature and allow for efficient composition in a way that respects all of their properties. Every logic operation, e.g. conjunction, disjunction, and negation, can easily be lifted to the level of ROBDDs. With that, their construction from logic formulas becomes a trivial task.

The uniqueness of subgraphs is easily achieved with a simple lookup. We will only create new nodes if they did not exist previously and use the existing ones otherwise. The function $unique$ denotes exactly this behaviour, both for terminal nodes as well as for internal nodes:

Definition 5 (Unique) *For any Boolean value $b \in \mathbb{B}$, let $f = unique(b)$ be the unique terminal node with $value(f) = b$. For any variable $x_i \in \mathbf{x}$ and any two BDDs g and h, let $f = unique(x_i, g, h)$ be the unique internal node with $var(f) = x_i$, $then(f) = g$, and $else(f) = h$.*

With $unique$ alone, the construction of constant Boolean functions is already possible. On the other hand, non-constant functions can be constructed with the conditional function ite, i.e. if-then-else, in a recursive fashion. Based on two, possibly constant, decision diagrams, f and g, and one variable x_i, ite constructs the decision diagram that evaluates to f if x_i is true, and to g otherwise.

Definition 6 (If-Then-Else) *For any variable $x_i \in \mathbf{x}$ and any two BDDs f and g,*

$$ite(x_i, f, g) := \begin{cases} f & \text{if } f = g \\ unique(x_i, f, g) & \text{otherwise.} \end{cases}$$

The *ite* operation can generally be used to construct BDDs but, as it is defined here, it respects only the uniqueness property of ROBDDs. In contrast to that, the variable order is not enforced by definition but rather remains the user's responsibility. Generalising the *ite* function to also enforce the correct variable order is possible and the respective variants do exist.

The two primitives, *unique* and *ite*, are sufficient to realise all logic operations directly on ROBDDs in a recursive fashion. Exploiting Boole's expansion theorem[2][52], we can *split* the given ROBDD at its smallest variable and proceed recursively. With *ite* the respective results can be reassembled, thus performing the logic operation in a property-preserving fashion. We define *apply* generically for all the binary Boolean operations. This subsumes, in particular, the operations for conjunction and disjunction, $\wedge$ and $\vee$.

Definition 7 (Apply) *Let $\langle op\rangle \in \mathbb{B}\times\mathbb{B} \longrightarrow \mathbb{B}$ be a logic operation and let f and g be two BDD operands. The definition of $\langle op\rangle$ generalises to BDDs with*

$$f\ \langle op\rangle\ g := \begin{cases} ite(x_i, f_t\ \langle op\rangle\ g_t, f_e\ \langle op\rangle\ g_e) & \textit{if } f, g \textit{ are int. nodes with } x_i = var(f) = var(g) \\ ite(x_i, f_t\ \langle op\rangle\ g, f_e\ \langle op\rangle\ g) & \textit{if } f \textit{ is internal node with } x_i = var(f) < var(g) \\ ite(x_i, f\ \langle op\rangle\ g_t, f\ \langle op\rangle\ g_e) & \textit{if } g \textit{ is internal node with } x_i = var(g) < var(f) \\ unique(f_v\ \langle op\rangle\ g_v) & \textit{if } f, g \textit{ are terminal nodes} \end{cases}$$

$$\begin{aligned} \textit{with } f_t &:= then(f) \\ f_e &:= else(f) \\ f_v &:= value(f) \\ \textit{and } g_v &:= value(g). \end{aligned}$$

In the very same way, monadic operations can be lifted to the level of ROBDDs. Although there are only the two monadic operations for negation $\neg$ and for the identity, we also formulate *monadic apply* in the most general way.

Definition 8 (Monadic Apply) *Let $\langle uop\rangle \in \mathbb{B} \longrightarrow \mathbb{B}$ be a monadic logic operation and let f be a BDD operand. The definition of $\langle uop\rangle$ generalises to BDDs with*

$$\langle uop\rangle\ f := \begin{cases} ite(x_i, \langle uop\rangle\ f_t, \langle uop\rangle\ f_e) & \textit{if } f \textit{ is internal node with } x_i = var(f) \\ unique(\langle uop\rangle\ f_v) & \textit{if } f \textit{ is terminal node.} \end{cases}$$

$$\begin{aligned} \textit{with } f_t &:= then(f) \\ f_e &:= else(f) \\ \textit{and } f_v &:= value(f) \end{aligned}$$

[2]Also referred to as Shannon expansion [51].

For brevity, we will, from here on, assume all decision diagrams (i) to be fully reduced and (ii) to conform to a strict variable order. We will use the terms BDD, OBDD, and ROBDD interchangeably.

ROBDDs build upon the standard Boolean logic $(\mathbb{B}, \wedge, \vee, \neg)$ but the definitions for *unique*, *ite*, and *apply* are really not exploiting any properties of $\mathbb{B}$. We will see that the underlying algebraic structure can, in fact, easily be replaced, allowing for the generalisation of Algebraic Decision Diagrams (ADDs).

2.3 Algebraic Decision Diagrams

While BDDs lift the standard Boolean logic to the level of decision diagrams, ADDs[3] do this for all other algebraic structures. The generalisation was originally introduced for matrix representation and manipulation [53, 54] but ADDs were soon adopted for a wider variety of algebraic structures [27, 28].

Definition 9 (Algebraic Structure) *An algebraic structure is a tuple*

$$\mathcal{A} := (\mathcal{S}, \langle op_1 \rangle, \langle op_2 \rangle, \ldots, \langle uop_1 \rangle, \langle uop_2 \rangle, \ldots, e_1, e_2, \ldots)$$

with a carrier set $\mathcal{S}$ and a finite number of binary operations $\langle op_1 \rangle, \ldots, \langle op_n \rangle \in \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$, monadic operations $\langle uop_1 \rangle, \ldots, \langle uop_m \rangle \in \mathcal{S} \rightarrow \mathcal{S}$, and distinguished elements $e_1, \ldots, e_l \in \mathcal{S}$.

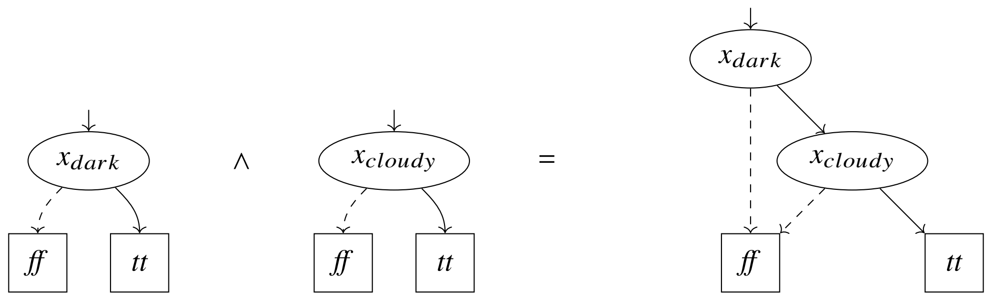
For any such structure, we can derive the corresponding decision diagrams, i.e. the respective ADDs. All of the definitions concerning BDDs, i.e Definition 1-8, and, most importantly, the main Theorem 1 naturally carry over to the new generalised data structure. For any algebraic structure $\mathcal{A}$, the corresponding ADDs form a canonical and size-optimal representation for functions of the form $\mathbb{B}^n \rightarrow \mathcal{A}$. The (monadic) operations of $\mathcal{A}$ translate to the corresponding (monadic) operations on ADDs and the distinguished elements translate to the corresponding constant ADDs.

As one class of algebraic structures, various logics seamlessly transfer to decision diagrams analogous to the standard Boolean logic. This allows us to loosen the hard binary nature of Boolean values to multi-valued logics, e.g. Kleene and Priest logics [55], and even to the various fuzzy logics [56]. The advantages of the derived ADDs are the same as that of their respective underlying logics: they allow to model missing knowledge and uncertainty in the represented functions.

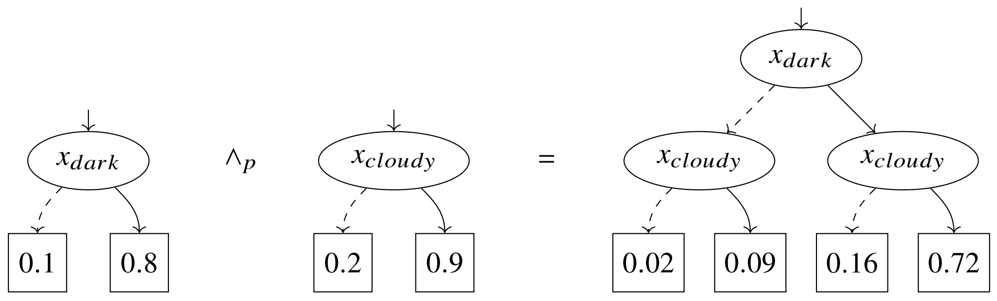
As just two representatives for the various fuzzy logics, consider the probabilistic and the min-max definition of conjunction, disjunction, and negation. Both logics operate on the interval $[0, 1]$.

Definition 10 (Probabilistic Fuzzy Logic) *The Probabilistic Fuzzy Logic* $\mathcal{A}_p = ([0, 1], \wedge_p, \vee_p, \neg_p)$

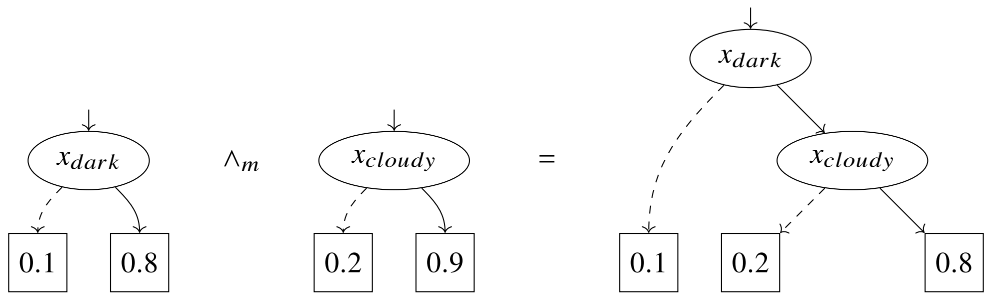
[3]Also called Multi-Terminal Binary Decision Diagrams (MTBDDs).



(a) Conjunction on BDDs.



(b) Conjunction on Probabilistic Logic ADDs.



(c) Conjunction on Min-Max Logic ADDs.

Figure 2.2: Example ADDs for the prediction of rainfall. In this example, the conjunction of two criteria yields the overall prediction.

is defined with

$$\begin{aligned} &\textit{conjunction} && a \wedge_p b := ab \\ &\textit{disjunction} && a \vee_p b := 1 - (1-a)(1-b) \\ &\textit{and negation} && \neg_p a := 1 - a. \end{aligned}$$

Definition 11 (Min-Max Fuzzy Logic) *The Min-Max Fuzzy Logic $\mathcal{A}_m = ([0,1], \wedge_m, \vee_m, \neg_m)$ is defined with*

$$\begin{aligned} &\textit{conjunction} && a \wedge_m b := min(a,b) \\ &\textit{disjunction} && a \vee_m b := max(a,b) \\ &\textit{and negation} && \neg_m a := 1 - a. \end{aligned}$$

Figure 2.2 illustrates the semantic differences between the ADDs over the three logics. In this example, we want to predict whether it will rain and we base this prediction on the two observations: if it is already dark and cloudy. Both criteria alone are indicators but only combined, they yield a good predictor for rainfall. In contrast to the BDDs (Fig. 2.2a), the two ADDs based on fuzzy logics (Fig. 2.2b and 2.2c) can weight the criteria and, in this way, model a notion of uncertainty. Both logics aim at the same problem but they yield very different behaviour which manifests itself in the different topologies of their respective ADDs.

ADDs are by no means limited to logics but they allow for any other algebraic structure in the very same fashion. In the course of this dissertation, we have used various ADDs with algebraic structures ranging from logics [6, 7]$_{\mathrm{AP}}$ and other lattice-like structures to monoids, groups, and fields [3, 4, 5]$_{\mathrm{AP}}$. The data structure is easily adoptable and generally facilitates the aggregation of functions — a great tool for program optimisation.

2.4 Other Variants of Decision Diagrams

The most important variant of decision diagrams in the context of this dissertation are undoubtedly ADDs. However, many more variants of the data structure have emerged, three of which we will discuss briefly.

Like ADDs are a generalisation of BDDs with regard to their co-domain, Multi-Value Decision Diagrams (MVDDs) [45] generalise the input domain and operate on variables with an arbitrary but finite input domains. Instead of the binary decisions at its internal nodes, a discrete input domain $\mathcal{D}_i$ per variable x_i allows for n-ary decision making. The properties of BDDs are, again, fully preserved making MVDDs a canonical representation for functions of the form $\mathcal{D}_0 \times \ldots \times \mathcal{D}_{n-1} \rightarrow \mathbb{B}$.

Not all variants of decision diagrams are generalisations of BDDs. The prime example for an adaptation are Zero-Suppressed Decision Diagrams (ZDDs) [43], a variant that uses a modified reduction rule. Instead of eliminating nodes whose two successors are equal, in ZDDs those nodes are considered redundant whose *then*-successor points to the 0-terminal. Although not obvious, also this adaptation yields a canonical representation for Boolean functions $\mathbb{B}^n \to \mathbb{B}$. ZDDs are particularly suited to encode sparse sets where they can drastically reduce the number of nodes needed, i.e. the data structure's memory footprint.

In some cases, it is clear whether BDDs or ZDDs are the suitable data structure. However, in other cases the choice might not be as obvious, yet its impact can be critical for the diagram's size. An adaptation that aims at exactly this problem are Chain-Reduced Decision Diagrams (CDDs) [44]. They combine the power of both reduction rules, those of BDDs and those of ZDDs, and yield a decision diagram without redundant nodes in both senses. This property comes at the cost of additional bookkeeping per node, i.e. what reduction rule to apply locally, making the data structure larger than its equivalent BDD and ZDD by no more than a constant factor.

3 (Domain-Specific) Program Optimisation with Algebraic Decision Diagrams

This dissertation explores the potential of Algebraic Decision Diagrams (ADDs) for the optimisation of *programs*. The term *programs* is most widely associated with those written in some general-purpose programming language like, e.g., C++, Python, or Java. However, the range of programming languages out there is much more diverse than these few extremely popular languages that everyone knows. With many language workbenches and frameworks [57, 58, 59, 60, 61, 62][63, 64, 65, 66] that allow for the rapid development of new languages, sometimes dedicated to a single purpose, the diversity of the programming languages landscape has steadily increased $[6]_{\mathrm{AP}}$.

In this dissertation, we focus on languages that can best benefit from decision diagram technology for their running time and size optimisation. These languages turn out to be quite diverse and they range from Domain-Specific Languages (DSLs) that are closely tied to decision diagrams, to less obviously suited inputs, and even to general-purpose programming languages.

In the classical compiler pipeline, an input program is first parsed and transformed into one, or sometimes multiple, Intermediate Representations (IRs). On this basis, the program is then optimised before the final result is emitted as executable code for some target platform. In this sense, we consistently use ADDs as the IR for our program optimisation. The optimisation aspect is, in part, obvious here because ADDs inherently optimise the represented function as discussed in Chapter 2. In addition, domain-specific optimisations can be applied on the bases of ADDs, just like they would be applied on any other IR.

The representative programming languages that we used to evaluate the potential of ADDs as an IR each fall into one of three categories:

- **Domain-Specific Languages (DSLs):** These languages are tailored to (i) their users' mindsets and here also (ii) with their potential for ADD-based optimisation in mind.

- **Machine-learned models:** Also machine-learned models can be considered a form of a domain-specific input language. It turns out that Random Forests are particularly suited for radical ADD-based aggregation and optimisation.

- **General-purpose programming languages:** Moving from domain-specific to general-purpose languages, we use the *while* language as a representative to explore a radically new optimisation paradigm — again based on ADDs as an IR.

3.1 Domain-Specific Languages

With ADDs being suited for a particular class of functions, namely $\mathbb{B}^n \rightarrow \mathcal{A}$, they are naturally suitable for DSLs that aim at expressing exactly this kind of function. These languages are therefore a prime example and a good starting point for ADD-based program optimisation. One could expect all of these languages to be structurally extremely similar as they are closely related to their IR, the ADDs. However, the contrary is the case: In the course of this dissertation, we will explore two quite different languages:

- **Graphical** decision diagram language: What is actually a small collection of graphical DSLs can be used to model decision services as acyclic graphs, compose them in a hierarchical fashion, and also to experiment with the semantics of different algebraic structures.

- **Rule-based** language: Quite different in its mindset, MiAamics [67][8]$_{\text{AP}}$ is a textual DSL for extremely large systems of conditional rules. It aggregates these rules into a single ADD and, in this way, partially evaluates them at compile time.

Graphical Domain-Specific Decision Diagram Languages

A DSL that closely matches the structure of ADDs can obviously greatly benefit from their inherent optimisation potential already at compile time. For this reason, it is the natural and promising starting point for ADD-based program optimisation that we will discuss in this section. Like every other DSL, also this language should capture its users mindset, in this case an ADD-centred way of thinking. The target group for such a language is probably not too large but, we found enthusiastic users during a summer school [7]$_{\text{AP}}$. In fact, the resulting graphical DSL was a great tool to experiment with ADDs and to quickly get a feeling for their capabilities and limitations in different contexts and based on varying algebraic structures. Many of the ideas in the more sophisticated ADD-based program optimisations (Sec. 3.1, 3.2, and 3.4) were built on top of this expertise.

Modern software development paradigms, such as Language-Driven Engineering (LDE) [6]$_{\text{AP}}$ and Language-Oriented Programming (LOP) [60], encourage exactly this development of, possibly single purpose, DSLs — paradigms that have become possible with the increasingly cheap development of new DSLs using modern language and meta-modelling frameworks [59, 68]. For our decision diagram language, we utilise Cinco [59], a meta-modelling framework that allows for the rapid development of graph-based DSLs. Cinco generates an entire mIDEs fully automatically in which the desired DSL can be used visually (Fig. 3.1).

Our decision diagram language provides a means for the creation of ADDs, or decision services as we call them when embedded in a larger software project. The result is always a decision function, may it be in the form of a visualisation or as generated program code. Its input domain

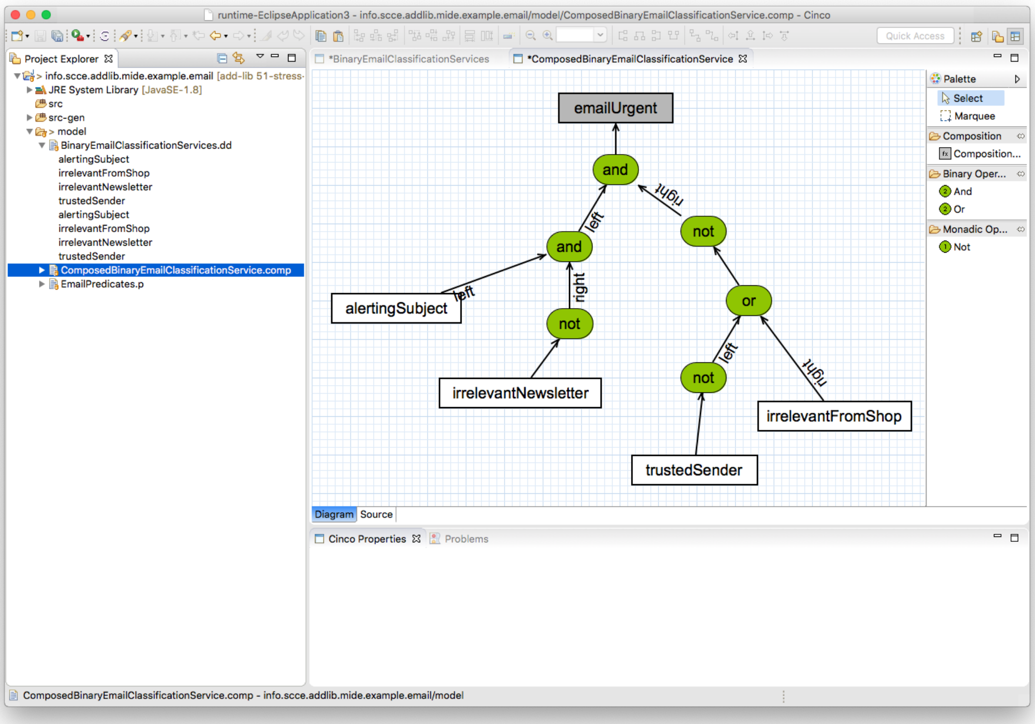
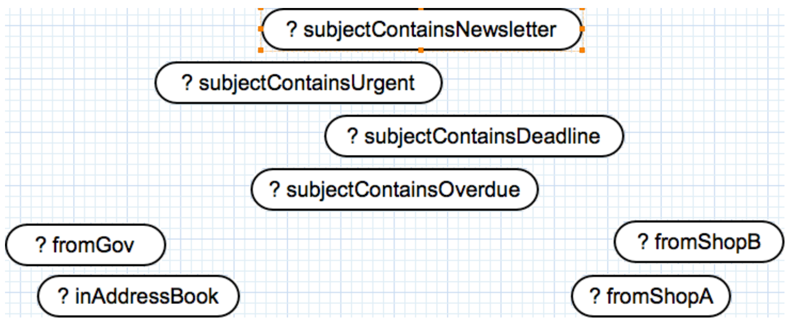
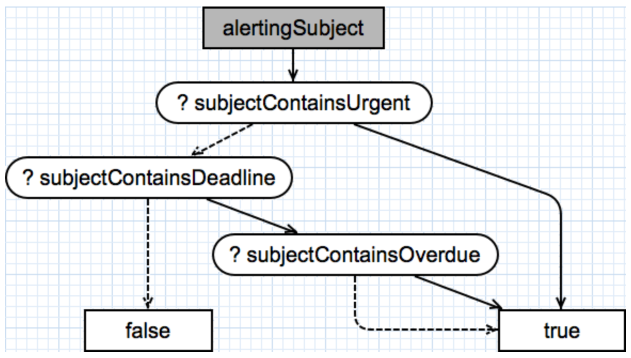

runtime-EclipseApplication3 - info.scce.addlib.mide.example.email/model/ComposedBinaryEmailClassificationService.comp - Cinco
Quick Access
Project Explorer
info.scce.addlib.mide.example.email [add-lib 51-stress-
JRE System Library [JavaSE-1.8]
src
src-gen
model
BinaryEmailClassificationServices.dd
alertingSubject
irrelevantFromShop
irrelevantNewsletter
trustedSender
alertingSubject
irrelevantFromShop
irrelevantNewsletter
trustedSender
ComposedBinaryEmailClassificationService.comp
EmailPredicates.p
*BinaryEmailClassificationServices
*ComposedBinaryEmailClassificationService
emailUrgent
and
left
right
and
not
alertingSubject
left
right
not
or
irrelevantNewsletter
left
not
right
irrelevantFromShop
trustedSender
Palette
Select
Marquee
Composition
Composition...
Binary Oper...
And
Or
Monadic Op...
Not
Diagram
Source
Cinco Properties
Problems
ComposedBinaryEmailClassificationService.comp - info.scce.addlib.mide.example.email/model

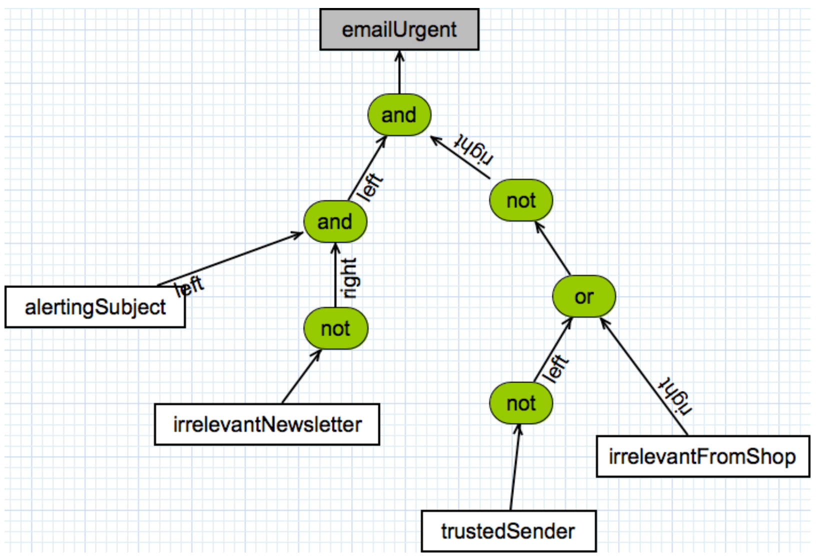
Figure 3.1: Screenshot of ADD composition DSL in its dedicated Mindset-Supporting Integrated Development Environment (mIDE).



(a) Predicate model.



(b) Exemplary decision diagram model.



(c) Composition model.

Figure 3.2: Exemplary models in the dedicated modelling languages.

is arbitrary, other than that of ADDs, but, also here, it must be discretised eventually. The DSL is actually a collection that consists of three sub languages, each referencing elements in the others:

- **Predicates** are used to describe the input domain in a way that translates seamlessly to our chosen form of representation, to ADDs. This is where the translation from the arbitrary input domain to that of the underlying ADDs is realised. The corresponding models (Fig. 3.2a) in the DSL are extremely simple as they provide no more than the later referenceable predicate names and links to their respective implementation if needed.

- **Decision diagrams** are the central kind of model (Fig. 3.2b) and they resemble the structure that we know from ADDs, however, with a few crucial differences. Other than the data structure, the DSL enforces neither (i) an order on predicates, nor (ii) canonicity of the modelled decision diagram. These crucial properties are later enforced fully automatically at compile time through corresponding model transformations.

- **Composition** models built on top of decision diagram models. They utilise the underlying algebraic structure and allow for the composition of decision diagrams (Fig. 3.2c). These models are structurally similar to syntax trees with the difference that they are actually Directed Acyclic Graphs (DAGs) that do allow for reuse.

We will sketch the expressiveness of this language collection with the help of an email classification example that was also used in [6]$_{\text{AP}}$ and [7]$_{\text{AP}}$. In this example, the aim is to automatically find emails that are of particular importance to a user — a task that is often encountered in email applications. The function to be realised takes an email as its input and produces, in its simplest form, a Boolean output.

In the first step, an incoming email is described by some predicates, each of which may be implemented in Java or some other general-purpose programming language. For an incoming email, the predicates seen in Figure 3.2a are only some of the indicators used in [6]$_{\text{AP}}$. An email that, for example, contains the string "Newsletter" already in its subject would usually not be of particular importance to any user. Based on this collection of predicates, more complex decision logic can be realised in the decision diagrams. The example rule in Figure 3.2b aims to detect emails whose subject line appears alerting. While all of these rules alone are probably not particularly strong indications for an important email, their composition can yield a powerful classification service. Figure 3.2c shows just one possible example of such a composition.

All of the models are valid input programs to our DSL and their structural proximity to ADDs allows for a direct transformation to the data structure. This step yields a canonical and fully reduced ADD as the IR of the compilation process. On this basis, the corresponding implementation in various general-purpose programming languages can be generated. An embedding software project, for example an email application, can then take advantage of this highly optimised implementation in a service-oriented fashion.

Just like ADDs, also our DSL allows for interchangeable algebraic structures. In fact, yet another DSL allows defining a custom algebraic structure on a meta-level [7]$_{\text{AP}}$. With its degree of flexibility — both on the meta-level with regard to the algebraic structure and on the modelling level with regard to the concrete decision diagrams — our decision diagram DSL has proven to be a great tool for the experimentation with ADDs. The small example of email classification may be performance critical only for very high throughputs. However, all of the subsequently discussed ADD-based optimisation techniques are in principle expressible with this collection of DSLs.

On a side note, many email applications realise the here discussed email classification in the form of a rule-based DSL. In this spirit, our second DSL, MiAamics, may be utilised to tackle the exact same problem in a completely different way.

MiAamics: A Rule-Based Domain-Specific Language

A second, quite different example for an ADD-based DSL is MiAamics, a textual specification language for recommender systems $[8]_{\mathrm{AP}}$. Based on some description of a given situation, MiAamics allows for the definition of the system's recommendation behaviour by means of a potentially large set of *recommender-style* rules. Each of these rules individually gives no more than a hint for a good decision but together the outcome can be quite powerful. In this way, the system can, e.g., recommend matching products based on a user's previous behaviour, the time of the day, or other environment variables.

The objective of MiAamics is to provide domain experts, who are well-versed with the semantics of recommender systems but may otherwise be unfamiliar with programming languages, with a specification means that suits their already established mindset. Rule-based realisations are indeed already used to define the behaviour of recommender systems [69, 70]. They allow domain experts to utilise their expertise and to define all behaviour in a declarative fashion, without having to worry about performance at all. The highly efficient implementation is then generated fully automatically by the MiAamics compiler.

This rule-based approach is a good example for the simplicity principle proposed in [71]. The allowed rules are not fully general, but they elegantly capture a bulk of decision problems for which they are much better suited than any general-purpose language. In previous experience, marketing and campaign managers with no ability to program nor familiarity with databases or query languages were perfectly able to master the MiAamics rule formats [33, 32].

The primary concern of MiAamics is, of course, to capture its users' mindset but it was also designed with its potential for optimisation in mind. The central question is how to transform a potentially large system of rules to an efficient representation and implementation that, ideally, saves both memory and evaluation time relative to the original system of rules.

We will sketch the MiAamics approach $[8]_{\mathrm{AP}}$ with the help of a small example: a recommender system that suggests an appropriate wine to serve with a given dish. In this example, the function's arguments describe the situation, i.e. the dish that is served, and its result is a wine suggestion. For simplicity, we characterise dishes by whether or not they contain *cheese*, *meat*, and *fish*. The wine, on the other hand, is described with the three attributes *red*, *white*, and *sweet*. With its small size, the example is nicely suited for illustration, however, the real optimisation potential comes into play for much larger collections of rules.

There are many rules for a good choice of wine, but some of them are more important than others. In this example we give high priority to serving *white* wine with every meal that contains *cheese* or *fish*. The recommender system can select from a set of possible recommendations, the so called target elements. For the wine example, let us consider three different wines, each characterised by a unique name and some Boolean attributes:

$$\begin{aligned}\mathcal{E} := \{ & (sauvignon\ blanc, \{ white \}),\\ & (riesling, \{ sweet, white \}),\\ & (merlot, \{ red \}) \}.\end{aligned}$$

Analogous to the definition of the possible outcomes $\mathcal{E}$, also the textual DSL requires the users to define these elements. The format is analogous to the mathematical definition but could also easily be realised with some graphical interface.

With the wine collection in place, we can now embed our wine expertise in MiAamics rules and, in a sense, teach the recommender system. Every rule consists of three parts:

- **Condition:** Based on the situation variables, a rule's condition describes the situations in which it is applicable in the form of a Boolean expression. For a given situation — here the description of a dish — the rule is applicable if its condition evaluates to true and we ignore the rule otherwise.
- **Selection:** The selection describes the affected target elements based on their Boolean attributes — here the wines from our collection. The selected ones are recommended by this particular rule but not yet by the overall recommender system.
- **Weight:** Because some rules may be more important than others, every rule is associated with a weight which is a real number that may even be negative. The higher the weight, the more this rule contributes to the overall recommendation.

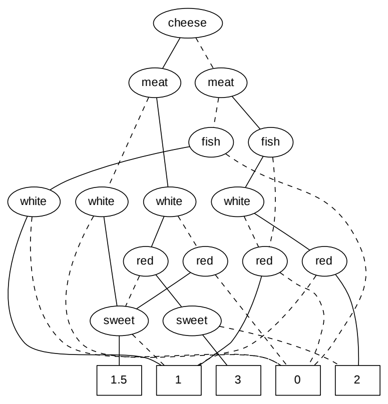
I am admittedly not an expert for wines but, with some advice, came up with a reasonable set of recommendation rules:

$$\begin{aligned}\mathcal{R} := \{ & \texttt{if}\ (cheese \lor fish)\ \texttt{add}\ 1.0\ \texttt{to}\ (white),\\ & \texttt{if}\ (meat)\ \texttt{add}\ 1.0\ \texttt{to}\ (red),\\ & \texttt{if}\ (cheese)\ \texttt{add}\ 0.5\ \texttt{to}\ (sweet \land white),\\ & \texttt{if}\ (meat \land cheese)\ \texttt{add}\ 0.5\ \texttt{to}\ (sweet \land red) \}.\end{aligned}$$

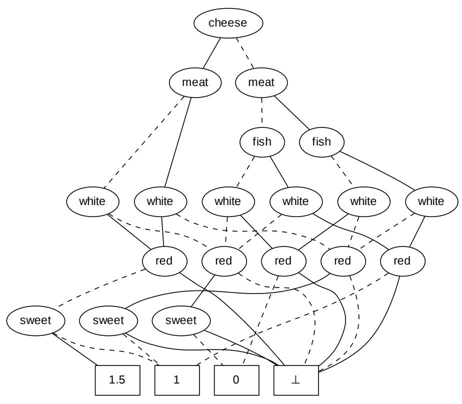
Most importantly, we ensure, with a high priority, to serve *white* wine with dishes that contain *cheese* or *fish*. In the very same way, we ensure that *red* wine is served together with *meat*.

There is an obvious way to evaluate these rules which is to process them one after the other and to accumulate the weights per target element, i.e. per wine. However, if we did that we would neglect the inherent optimisation potential of these rules. This is where the strength of the MiAamics approach comes into play: With partial evaluation of the given rules $\mathcal{R}$ and under consideration of the available target elements $\mathcal{E}$, it is possible to drastically speed up the overall evaluation process.

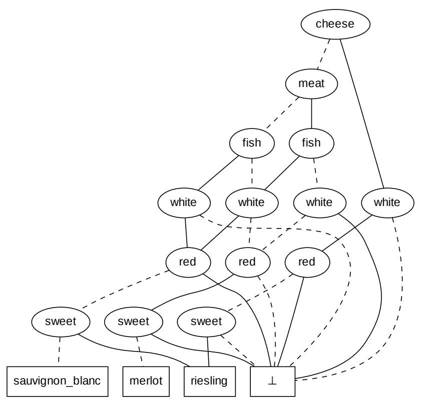
The key technique to partial evaluation are, again, ADDs and MiAamics utilises multiple kinds with varying underlying algebraic structures. The original rules are step-wise transformed to intermediate ADDs before the final optimised representation is derived.



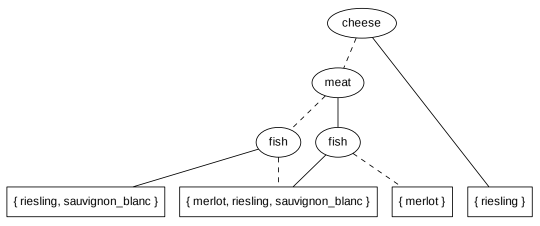
(a) Aggregated rule ADD.



(b) Filtered rule ADD with existing elements only.



(c) Filtered rule ADD with highest-scoring elements only.


cheese
meat
fish
fish
{ riesling, sauvignon_blanc }
{ merlot, riesling, sauvignon_blanc }
{ merlot }
{ riesling }

(d) Final decision ADD.

Figure 3.3: The MiAamics pipeline step-wise transforming the system of rules into a single decision ADD.

In the first step, every rule in $\mathcal{R}$ is transformed to its own ADD over the real numbers $(\mathbb{R}, +, 0)$. These diagrams essentially represent the conjunction of their respective rule's condition and selection expression. If this conjunction holds, the ADD yields the rule's weight and 0, the neutral element of summation, otherwise. The obtained collection of ADDs allows for natural composition by summation and thus aggregates the potentially large set of rules $\mathcal{R}$ into a single ADD. Figure 3.3a shows this aggregated rule ADD for our running example. The obtained function yields the accumulated weight for a given description of the situation and for one particular target element, i.e. for a given dish and wine description. The structure is, however, far from performance-friendly: For its evaluation at runtime only the situation description, i.e. the description of a dish, is known. Finding the highest-scoring target elements, i.e. the best wines, on the other hand, requires one iteration over all of the possibilities in $\mathcal{E}$.

To resolve this shortcoming three subsequent transformations are necessary all of which are pure ADD transformations. Filtering the ADD for those target elements that actually exist in $\mathcal{E}$ eliminates the necessity to iterate over $\mathcal{E}$. The resulting ADD is shown in Figure 3.3b. Its function is the exact same as that of the previously aggregated ADD except for those target elements that do not exist in $\mathcal{E}$. These cases are now marked with the dedicated $\perp$ which simplifies subsequent transformations. It is for the newly introduced $\perp$ symbol, that this second stage in the optimisation pipeline operates on ADDs over the accordingly extended real numbers $(\mathbb{R} \cup \{ \perp \}, +, 0)$.

In the final decision structure, we are no longer interested in the accumulated weight, but rather in the highest-scoring target element, i.e. the best wine. For this reason, the third step in the transformation pipeline maps the terminal values of all paths that lead to a highest score in their respective situation to their corresponding target element. All other paths that are not associated with a target element are merged into the $\perp$ terminal. The resulting ADD represents the function that still depends on situation and target variables, however, if a target element is reached, it is actually a highest-scoring target element. Figure 3.3c shows the corresponding ADD for our running example where the recommended wines are now held directly in the terminal nodes. At this stage, the ADDs are again over a new algebraic structure with the target elements $\mathcal{E}$ as its carrier set.

The remaining indirection in the evaluation process are the target variables that, until now, remain in the ADD. When evaluating the structure for a given situation, i.e. a description of a dish, we can simply trace down the top-half of the ADD. With respect to target variables, however, we are interested in all reachable target elements, i.e. all of the present highest-scoring elements. Again, we can collect these target elements effectively with simple ADD transformation at compile time. The resulting ADD holds sets of highest-scoring elements in its terminal nodes and depends solely on situation variables. This final ADD is associated with yet another algebraic structure, in this case a power set lattice $\mathcal{P}(\mathcal{E})$.

Figure 3.3d shows the final aggregated ADD. Already for this extremely small example, it is obvious that its evaluation requires fewer steps than the original collection of rules $\mathcal{R}$. The diagram depends only on the situation variables and, per the properties of ADDs, ensures that each of them is evaluated at most once. For a given description of a dish, we can trace down the diagram starting at its root until a set of recommended wines is reached. For example, a *fish* dish is ideally served with *riesling* or *sauvignon blanc* according to our collection of recommendation rules.

The MiAamics optimisation process is presented in a slightly simplified variant here. For more details on the four transformations and for the different kinds of ADDs involved in the process, please refer to $[8]_{\mathrm{AP}}$.

The speedup that we achieved with this approach is impressive. In our experiment, we aggregated more than $200,000$ randomly generated rules each reasoning over 5 situation variables and 5 target variables. A naive evaluation of these rules would consider every rule's condition and selection criterion separately to accumulate the weights per target element. With the aggregated ADD, on the other hand, we reach a final decision with no more than 5 steps through the data structure. In our experience, the MiAamics is limited only by the number of variables used to describe the situation and the target element. On the other hand, it scales extremely well with the number of rules. The number of rules in our experiments was chosen arbitrarily and we could have easily continued to millions.

The random source of these rules arguably induces a bias in the experiment but the results are nevertheless drastic speedups that motivate further investigation of the underlying potential. Although not exactly MiAamics, but nevertheless highly influenced, we will see how these ideas lead to impressive speedups in a real-world application with Random Forests (Sec. 3.2).

The MiAamics rule-based DSL is naturally suited for recommender systems as they are prominently used in online shops [72, 73] or on video streaming platforms [74, 75]. Even in the context of smart cities "recommendation techniques become essential tools assisting consumers in seeking the best available services" [76]. In fact, the MiAamics approach was first explored, prior to this dissertation, in this context personalisation of advertisement and offers in online shops [33, 32]. The then award-winning approach was developed from 2000 to 2001 by METAFrame Technologies GmbH, meanwhile patented [9], and also used to solve the Semantic Web Service Challenge problems [77, 67]. Building on the ideas of this earlier implementation, the new realisation adds the final transformation steps in the optimisation pipeline. Rather than a description of the desired outcome, we can retrieve the selected elements directly at runtime and with no further indirection.

The range of applications for this optimisation approach is by no means limited to recommender systems. In fact, any system that reasons over an input domain that can be characterised in the form of predicates and that produces a discrete outcome can be modelled with MiAamics-style rules. One particularly impressive example for this is the transformation of Random Forests into a semantically equivalent ADD which will be discussed in the following section.

3.2 A Machine-Learned Program: The Random Forest

A particularly impressive result of this dissertation is the ADD-based aggregation of Random Forests $[3, 1, 2]_{\mathrm{AP}}$, a widely-known and popular classifier in machine learning. The aim here is to optimise the running time as well as the size of the machine-learned models which, in this case, allows for impressive speedups and reductions in size by multiple orders of magnitude.

In contrast to ADD-based optimisation for DSLs, here, we apply the same underlying ideas to a widely-known and well-established use case rather than creating a new language with ADDs already in mind. The results show that the specific properties of Random Forests allow for a

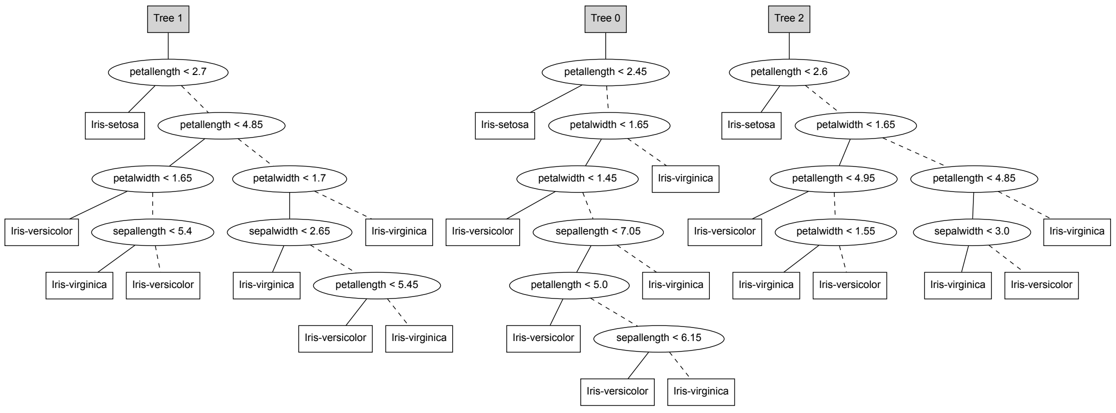


Figure 3.4: Random Forest learned from the Iris dataset [89].

tailor-made optimisation of the implicit program, beyond what a general-purpose compiler could possibly resemble. In this sense, Random Forests can be seen as their own very focused DSL that allows them to express nothing but collections of decision trees.

The optimisation of a Random Forest's running time is not a new problem but was, in fact, addressed multiple times, though with moderate success in the past. The simplest approaches generate code rather straightforwardly [78, 79][80, 81] and benefit mostly from native performance when compared to an in-memory interpretation. Other approaches achieve greater performance impacts through model simplification, however, this comes at the obvious cost of an undesired semantic change [82]. The aggregation of Random Forests, or more generally machine-learned models, was also tackled with semantic aggregation [83, 82, 84, 85]. These approaches, however, focus on a semantic aggregation and generally neglect the final running time and model size entirely — both criteria they were not aimed at in the first place. Also the reduction in size is a recognised problem that was addressed previously and with limited success [86, 87] — an approach interestingly sharing some similarities with the common ADD reduction techniques.

The only paper on Random Forests we know of that uses decision diagrams similar to our ADDs is [88]. However, they use these diagrams only to compact the individual tree and not to aggregate an entire Random Forest. In fact, the reported speedup by a factor of up to 61 seems more to rely on technical and even hardware details than on the use of decision diagrams. In contrast to that, our approach focuses on a holistic aggregation of entire Random Forests which, due to its globality, has a much greater impact. In fact, we obtain speed-ups already at the hardware-independent level that are orders of magnitude higher than those reported in [88].

All of these results, both for running time and size optimisation, are comparably small improvements when compared to the results that we achieve with our ADD-based aggregation.

For tangibility, we will discuss Random Forests and their transformation into an equivalent ADD with the help of a running example: Figure 3.4 shows a small Random Forests consisting of only three trees that were learned from the popular Iris flower dataset [89]. The dataset lists dimensions of Iris flowers' sepals and petals for three different species. Based on the given flower measurements, the task is to determine the corresponding flower species correctly. The problem at hand is a typical classification problem and can be approached with many classification methods in the field of machine learning — one of which are Random Forests. The algorithm is relatively simple and yields good results for many real-world applications. Its decision model generalises the training dataset that holds examples of input data labelled with the desired Iris species, also called *class*.

As its name suggests, a Random Forest consists of some number of decision trees, in this case three. Each of these trees is itself a classifier that was learned from a random sample of the training dataset. As a consequence, all trees are different in structure, they represent different decision functions, and can yield different decisions for the very same input measurements.

To apply a Random Forest to previously unseen input data, every decision tree is evaluated separately: Tracing the trees from their root down to one of the leaves yields one decision per tree, i.e. the predicted class or, in this case, the Iris species. The overall decision of the Random Forest is then derived as the most frequently chosen class, an aggregation commonly referred to as *majority vote*. Key advantage of this approach, compared to single decision trees, is the reduced variance. A more detailed introduction to Random Forests, decision trees, and their learning procedures can be found in [90, 91, 92].

When evaluation is performed in a per tree fashion, it is obvious that the computational effort involved grows linearly with the number of decision trees, i.e. the size of the Random Forest. While our illustrative example consists of only three trees there is essentially no limit to the number of trees. In fact, increasing its size can only improve the Random Forest and will, in contrast to other classifiers, not lead to overfitting [91].

The question arises if and how we can reduce the running time of this evaluation procedure to allow for a greater number of trees with no additional cost in running time. The solution to the problem requires a suitable representation that allows for effective program optimisation. Other than direct code generation approaches that use no IR at all, we have explored the potential of ADDs in this context. With their (i) predicates and binary branching at internal nodes and (ii) results on at leaf or terminal nodes, the Random Forests' individual trees share two important characteristics with ADDs, making them a promising candidate for ADD-based optimisation. It turns out that they not only allow us to seamlessly transform individual trees but their algebraic nature allows us to aggregate the entire collection of trees effectively. As a result the Random Forest is partially evaluated at compile time which often leads to a much smaller and faster to evaluate program structures.

The suitable kind of ADDs must represent the same decisions that the original Random Forest yields. The most conservative choice here is an ordered sequence of class labels, one for each tree. This sequence preserves all information and remains independent of any particular aggregation method, e.g. *majority vote*.

To formalise this, let C be the set of classes, e.g. the three Iris flower species. An individual tree's decision is one class $c \in C$, a Random Forest's decision is a word over these classes $\mathbf{c} \in C^*$. Note that we can describe the results of any Random Forests this way, no matter its size. In particular, we can represent the decision made by the empty Random Forest with the empty word ϵ and the results of a single decision tree with a word of length one. Moreover, this representation naturally allows for composition: we can simply concatenate the results of two distinct Random Forests, maintaining a one-to-one association between the word's symbols, i.e. the classes, and the corresponding tree in the original forest. With that in mind, we can define the algebraic structure *class words* as a monoid:

Definition 12 (Class Word Monoid) *Let C be the set of class labels. The class word monoid $\mathcal{W} = (C^*, \circ, \epsilon)$ is then defined with concatenation $\circ$ and the empty word ϵ.*

With the definition of the algebraic structure (Def. 12) we have also implicitly defined the corresponding ADDs which allow to represent functions that are very similar to that of the original Random Forest. The only difference is that the original Random Forest takes continuous features as its input while the ADD reasons over already evaluated predicates, i.e. its Boolean variables. Assuming an a-priori evaluation of all predicates for now, we already know that the Random Forest can be represented as an equivalent ADD.

The transformation of a given Random Forest to its equivalent ADD remains to be defined. In this process, we must guarantee the unique properties of decision diagrams:

- they enforce an order of predicates along all paths, and
- they are directed acyclic graphs that share common substructures where possible.

The natural way to achieve this is to rely solely on primitive ADDs, operations on them, and the ite operation. All of the corresponding algorithms ensure the invariant properties of ADDs (Chap. 2).

With the compositionality of the algebraic structure $\mathcal{W}$ (Def. 12) and the corresponding ADDs, we can transform any Random Forest incrementally. Starting with the empty Random Forest, we consider one tree after the other, aggregating a growing sequence of decision trees until the entire forest is entailed in the new decision diagram. We will first find a semantically equivalent decision diagram for the empty Random Forest, the neutral element of this aggregation procedure. Subsequently, we will describe a semantics-preserving transformation for single decision trees and a join operation to incorporate these decision diagrams into the overall aggregation.

No matter the input it was given, the empty Random Forest with 0 trees can only result in one outcome: the empty word ϵ. Hence, it resembles the constant function, also denoted ϵ for brevity, that is semantically equivalent to the constant decision diagram with ϵ as its only terminal node. This diagram forms the neutral element of our aggregation procedure.

To transform a single decision tree, we can build upon the well-known ADD construction operation ite (Chap. 2). For a predicate p and two decision diagrams f and g, $ite(p, f, g)$ constructs the diagram that evaluates to f if p holds and to g otherwise. We derive the decision diagram recursively along the tree structure, effectively delegating the entire transformation to well-known and efficient algorithms in a service-oriented fashion. The algorithm implementing ite ensures a strict predicate order and automatically shares substructures where possible. In fact, the resulting decision diagram is a canonical representation of the function for a given predicate order. Formally, this defines a function d_W mapping decision trees to decision diagrams over *class words* $w \in C^*$:

$$d_W(t) := \begin{cases} t_{val} & \text{if } t \text{ is leaf,} \\ ite(t_{pred}, d_W(t_{then}), d_W(t_{else})) & \text{otherwise.} \end{cases}$$

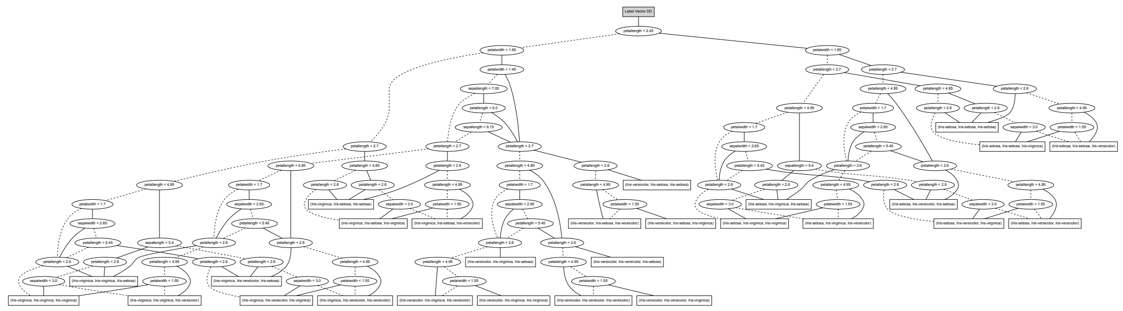


Figure 3.5: Partially evaluated Random Forest.

Having transformed every decision tree individually leaves us with the task to compose the resulting sequence of decision diagrams. This is where the above-mentioned compositionality of the class word monoid $\mathcal{W}$ (Def. 12) comes into play: Analogous to concatenation $\circ$ of class words, we can also apply concatenation to the derived ADDs. To ease readability, we also denote this terminal-wise concatenation of decision diagrams with the same symbol $\circ$.

The desired decision diagram aggregating an entire sequence of decision trees $t_0, t_1, \ldots, t_{n-1}$, can now be defined as

$$d_W(t_0, t_1, \ldots, t_{n-1}) := d_W(t_0) \circ d_W(t_1) \circ \cdots \circ d_W(t_{n-1}).$$

Figure 3.5 shows the aggregation of our example Random Forest (Fig. 3.4). Already, for this extremely small example, the average running time for classification is reduced. Its true impact, however, becomes apparent with increasing forest size.

Being the direct representation of the Random Forest's outcome, class words faithfully represent the information about the decisions of each individual tree. However, when the aim is to determine the most frequent class, i.e. the majority vote, this is far more than necessary. To this aim, only the frequency of each class is needed. The algebraic structure to represent this elegantly is the *class vector* monoid, where each component represents one class and its value the frequency with which that class was chosen.

Definition 13 (Class Vector Monoid) *Let C be the set of class labels. The class vector monoid $\mathcal{V} = (\mathbb{N}^{|C|}, +, \mathbf{0})$ is then defined with summation $+$ and the neutral vector $\mathbf{0}$.*

Based on this structure we can replay the development of the previous section by replacing $\mathcal{W}$ by $\mathcal{V}$, $\circ$ by $+$, and ϵ by $\mathbf{0}$. Indexing the class vectors directly with the class labels $c \in C$ rather than integers, this reads as follows:

The required representation of the empty decision is again provided by the neutral element, here the $\mathbf{0}$ vector, and a single class $c \in C$ can be represented naturally by a vector $\mathbf{i}(c)$ that is 0 everywhere except for position c, where it is 1. Also the construction of the corresponding ADDs with class vectors as their terminal values can simply be transcribed, as can the new transformation function d_V which differs only in its mapping from the tree's leaves to the new

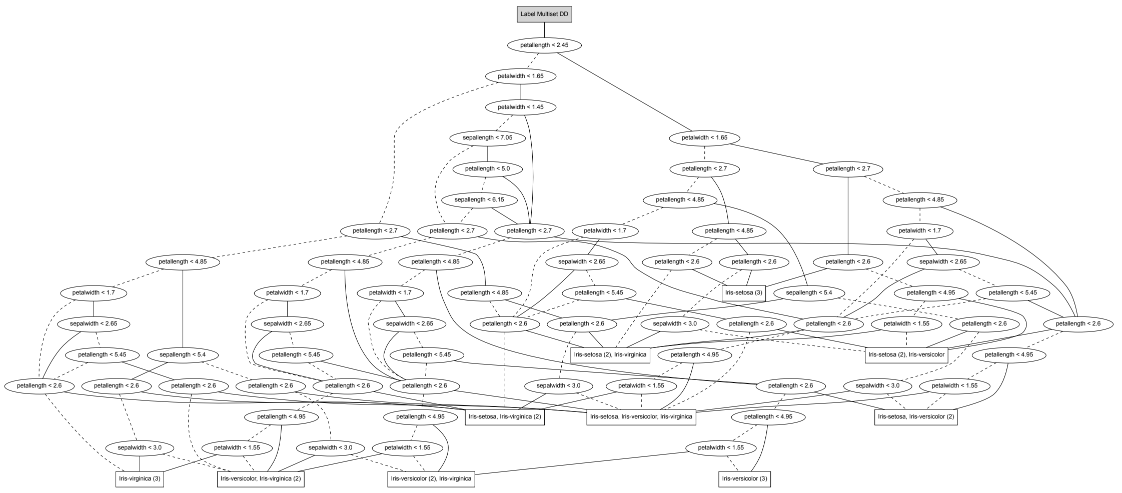

Label Multiset DD
Iris-setosa (3)
Iris-setosa (2), Iris-virginica
Iris-setosa (2), Iris-versicolor
Iris-setosa, Iris-virginica (2)
Iris-setosa, Iris-versicolor, Iris-virginica
Iris-setosa, Iris-versicolor (2)
Iris-virginica (3)
Iris-versicolor, Iris-virginica (2)
Iris-versicolor (2), Iris-virginica
Iris-versicolor (3)

Figure 3.6: Class vector abstraction of aggregated Random Forest.

carrier set:

$$d_V(t) := \begin{cases} \mathbf{i}(t_{val}) & \text{if } t \text{ is leaf,} \\ ite(t_{pred}, d_V(t_{then}), d_V(t_{else})) & \text{otherwise.} \end{cases}$$

Having adopted the underlying algebraic structure, all operations are seamlessly applicable to the corresponding ADDs as well. In this case, vector summation + is lifted to the new decision diagrams and we can again easily aggregate the Random Forest incrementally:

$$d_V(t_0, t_1, \ldots, t_{n-1}) := \sum_{i=1}^{n-1} d_V(t_i).$$

The new transformation abstracts from the order of class labels but maintains all the information required to construct and aggregate decision diagrams incrementally.

Abstracting from the order of class labels has two important advantages:

1. **Size** is reduced as many leaf nodes that differed only in the order of class labels are now unified. In fact, this effect can ripple up the entire decision diagram, i.e. the structure can partially collapse. Moreover, the vector representation itself also becomes more compact.

2. **Running time** for classification based on the new ADD is reduced as a result of the partial collapse of the structure: Where a predicate was previously needed to differentiate between two class words that differed only in the order of their class labels, this evaluation step becomes redundant. Moreover, the final aggregation step reduces to finding the maximal component of a single class vector.

Figure 3.6 shows the result of the class frequency abstraction for our running example.

The question arises if we can push the idea of abstraction further, i.e. if there exists an algebraic structure that abstracts from $\mathcal{V}$ and, at the same time, maintains its compositionality. It turns out

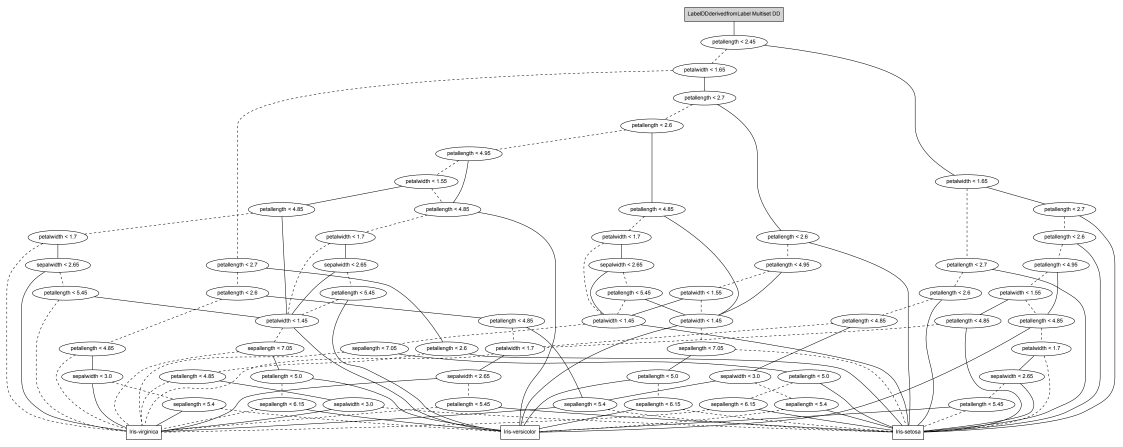


Figure 3.7: Most frequent label abstraction of aggregated Random Forest (majority vote).

that maintaining only the result of the majority votes violates the compositionality requirement. In fact, knowing the result of the majority votes for two distinct Random Forest gives no clue about the majority vote of the combined forest. Thus the most frequent class abstraction can only be applied at the very end, after the entire aggregation has been computed compositionally. The class frequency abstraction based on class vectors $\mathcal{V}$ provides the most concise compositional abstraction. Any further reduction directly leads to potential compositionality violations or, as we say, the class frequency abstraction is *fully abstract* for this scenario. Thus taking the formerly defined model transformation d_V to iteratively aggregate the trees of a Random Forest is provably the best we can do.

Only the subsequent monadic transformation mv that transforms a given class vector ADD to a pre-evaluated class ADD, holding the majority vote already in its terminals, remains to be defined. With ADDs, we can define this monadic transformation simply on the carrier set, i.e. on the class vectors. For any class vector $\mathbf{v} \in \mathbb{N}^{|C|}$ the majority vote is defined as

$$mv(\mathbf{v}) := \arg\max_{c \in C} \mathbf{v}_c.$$

Note that mv does not project into the same carrier set but rather from one algebraic structure $\mathcal{V}$ into another C. However, these transformations can be applied to the corresponding ADDs in the very same way as other operations can be. We can therefore define the final transformation of a Random Forest with its decision trees $\mathbf{t} = t_0, t_1, \ldots, t_{n-1}$ as

$$d_C(\mathbf{t}) := mv(d_V(\mathbf{t})).$$

Post-processing vector ADDs in this way has, again, quite an effect: Both size and running time are reduced for the same reasons as before, however, the coarser abstraction leads to stronger reductions. Moreover, the aggregation step for determining the final decision of the Random Forest, the majority vote, is no longer necessary.

Fig. 3.7 shows the result of the most frequent class abstraction for our running example.

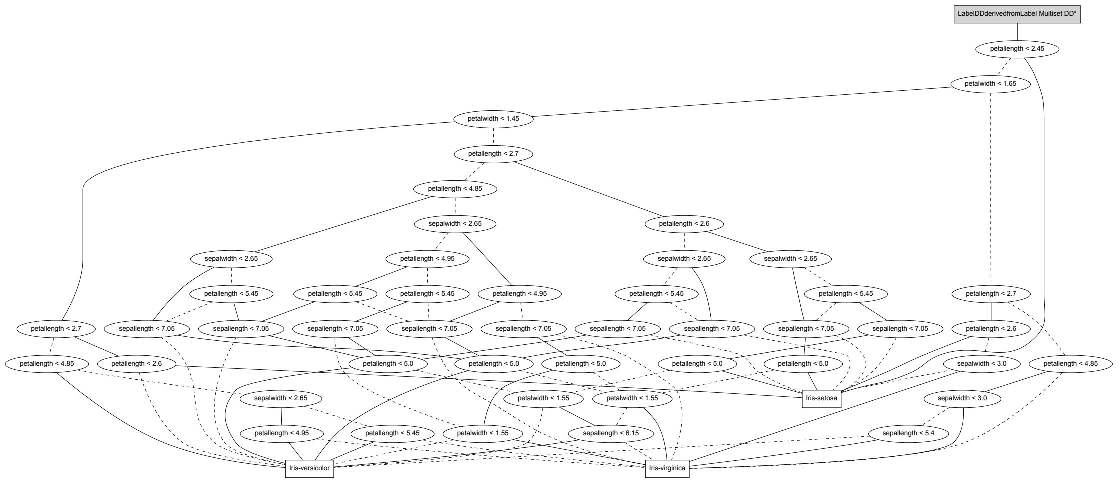


Figure 3.8: Most frequent label abstraction of aggregated Random Forest (majority vote) without semantically redundant nodes.

It turns out that the transformations discussed so far are not yet sufficient to guarantee true scalability, and this is despite the fact that they are optimal. The reason for this shortcoming is that all predicates are interpreted symbolically and their semantics are not taken into account.

When aggregating the trees of a Random Forest they all use varying sets of predicates. In contrast to simple Boolean variables, predicates are not independent of one another, i.e. evaluation of one predicate may yield some degree of knowledge about other predicates. For example, the predicate $petallength < 2.45$ induces knowledge about other predicates that reason over $petallength$: When the petal length is smaller than 2.45 it cannot possibly be greater or equal to 2.7 at the same time. This interplay is not taken care of by the symbolic treatment of predicates we followed until now.

Unsatisfiable path elimination, as illustrated by the difference between Figure 3.7 and Figure 3.8 for our running example, leverages the potential of a semantic treatment of predicates with significant effect:

- The **size** of decision diagrams is drastically reduced, and even
- the **running times** for classification further improves because semantically redundant decisions are eliminated.

Unsatisfiable path elimination depends on previous powerful abstraction: The trees in the original Random Forest have no unfeasible paths by construction. They are introduced in the course of our *symbolic* aggregation, which is insensitive to semantic properties.

The compositionality of the Random Forest transformation is fully preserved. Unsatisfiable path elimination can be applied at any time in the process and, in particular, during the stepwise transformation, before the final most frequent label abstraction, and also at the very end. This avoids that intermediate decision diagrams grow too large which would inhibit the scalability.

In contrast to the previous transformations, the elimination of unsatisfiable paths is not deterministic and, as a consequence, normal forms are no longer guaranteed. Thus our approach may yield different decision diagrams depending on the order of tree aggregation. It is, however, guaranteed that the resulting decision diagrams are minimal.

Unsatisfiable path elimination is a hard problem in general but in the here considered cases it is polynomial. [1] Our corresponding implementation uses SMT solving to eliminate all unsatisfiable paths.

All transformations build upon ite and algebraic operations on ADDs, all well-known procedures on the data structure. From the correctness of these primitives follows also the correctness of the here proposed transformations. The corresponding proofs are straightforward and essentially delegate the correctness argument to correctness proofs of the ADD primitives. Specific are here only the considered algebraic structures:

- class words (Def. 12) with concatenation to faithfully resemble the individual outcomes of the votes of the individual decision trees,
- class vectors (Def. 13) with addition to aggregate the outcome to an accumulated frequency per class, and
- the maximum frequency operation to precompute the majority vote.

A heuristic factor remains, the choice of predicate ordering, however, it impacts only the size of the resulting ADDs. We rely on the established technology implemented in the ADD-Lib [93] and in CUDD [94] to obtain good results also in this sense. Please note, that all transformations are semantics-preserving independent from the chosen ordering.

Theorem 2 (Correctness of Random Forest Transformation) *The transformations of a Random Forest to a class word ADD, to a class vector ADD, and to a class ADD are semantics-preserving.*

Unsatisfiable path elimination is its own field of research and, in contrast to the transformations above, there exist no normal forms in these cases. This is due to the fact that infeasible paths are treated as *don't cares*, and it is well-known that such a treatment does not lead to normal forms.

On the other hand, it is straightforward to step-wisely check the satisfiability of a path whenever the individual steps, i.e. the predicates, are decidable. This is obvious for the relation-based class of predicate considered here and for Random Forests in general. As each normal form has only finitely many paths, eliminating all unsatisfiable paths can be done effectively. In fact, this analysis is not a bottleneck of our construction.

Thus the correctness of unfeasible paths reduction only relies on the correctness of the used SMT solver for, in this case, quite simple predicates.

Theorem 3 (Correctness of Unsatisfiable Path Elimination) *The transformations of a Random Forest to a class word ADD, to a class vector ADD, and to a class ADD with subsequent elimination of unsatisfiable paths are semantics-preserving and every path in the resulting diagrams is feasible.*

[1]There are of course other theories for which it becomes an exponentially hard or even undecidable problem.

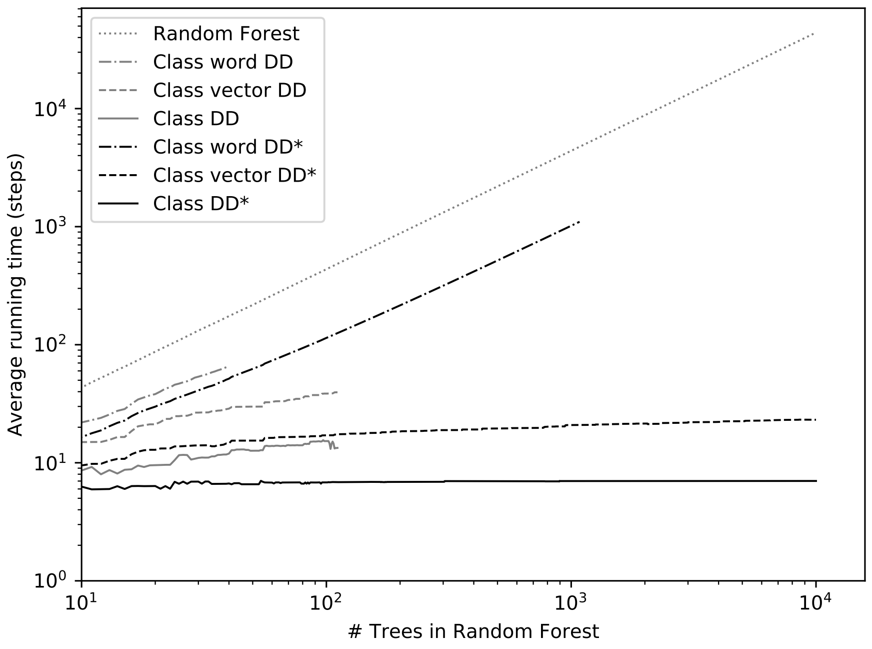


Figure 3.9: Average running time for classification over all examples in the Iris dataset [89].

Verifying the underlying ADD implementation is clearly a topic on its own. We can, however, check the correctness of the transformations via exhaustive testing:

- In the purely discrete case, we can simply compare the results for all possible cases in the original Random Forest with those in the derived ADD.
- In the continuous case, like in the running Iris example, we can semantically partition the input space according the distinguishing power of the involved predicate thresholds: In this case we compared the results for inputs covering all feasible paths in the derived ADD.

In this way, we can verify that the implementation worked correctly in all cases that we report in our experiments.

The three tree accompanying example is useful to explain the concepts but inadequate to illustrate the impact of our radical aggregation. For the same Iris dataset, we can build much larger Random Forests and compare their size and running times for the various transformations discussed (Fig. 3.9 and 3.10). In addition to that, we analysed the approach on various other datasets from the UCI Machine Learning Repository [95] (Tab. 3.1 and 3.2). All the reported classification time and size results were determined as the average over the entire corresponding data sets. For the Iris flower example these are 150 records, a number also explaining the smooth result graphs.

Our implementation relies on the standard Random Forest implementation in Weka [80] and on the ADD implementation of the ADD-Lib [93]. Please note that the considered datasets have

100		
Dataset	**Random Forest**	**Final ADD**
Balance Scale	802.21	7.71 (-99.04%)
Breast Cancer	1,298.72	17.12 (-98.68%)
Lenses	452.50	3.67 (-99.19%)
Iris	436.11	6.82 (-98.44%)
Tic-Tac-Toe	1,066.66	14.25 (-98.66%)
Vote	693.57	9.02 (-98.70%)

1,000		
Dataset	**Random Forest**	**Final ADD**
Balance Scale	8,014.12	7.73 (-99.90%)
Breast Cancer	13,020.03	17.11 (-99.87%)
Lenses	4,431.42	3.67 (-99.92%)
Iris	4,395.77	6.97 (-99.84%)
Tic-Tac-Toe	10,733.68	14.22 (-99.87%)
Vote	6,921.56	8.33 (-99.88%)

10,000		
Dataset	**Random Forest**	**Final ADD**
Balance Scale	80,277.03	8.16 (-99.99%)
Breast Cancer	130,361.20	17.73 (-99.99%)
Lenses	43,883.79	3.67 (-99.99%)
Iris	44,043.89	7.01 (-99.98%)
Tic-Tac-Toe	107,300.69	14.18 (-99.99%)
Vote	69,216.62	8.30 (-99.99%)

Table 3.1: Running time improvements for classification with Random Forests of size 10.000 for other datasets [95].

been developed with evaluations of this kind in mind, by independent parties, and that we are not using any additional data for our transformation. Thus our analysis can be considered unbiased.

Optimising the classification time is the primary goal of our approach. As wall clock time measurements are very sensitive to implementation details and machine profiles, we decided for the, in our eyes, more objective measure of step count for performance analysis. As steps we consider here the steps through the corresponding data structures, and in cases where the most frequent class must be computed at runtime, we account for one additional step per read. For both, the original Random Forest and the class word-based decision diagram these are n additional steps and the class vector variant needs $|C|$ additional steps.

Figure 3.9 shows the average evaluation times of the decision models for Random Forests of up to $10,000$ trees. The evaluation time of the original Random Forest grows linearly as expected: every new tree contributes approximately the same running time. Due to the large number of trees relative to their individual sizes our measurements appear as an almost straight line.

Already, the class word-based diagrams (see Class word DD in Fig. 3.9) reduce the classifica-

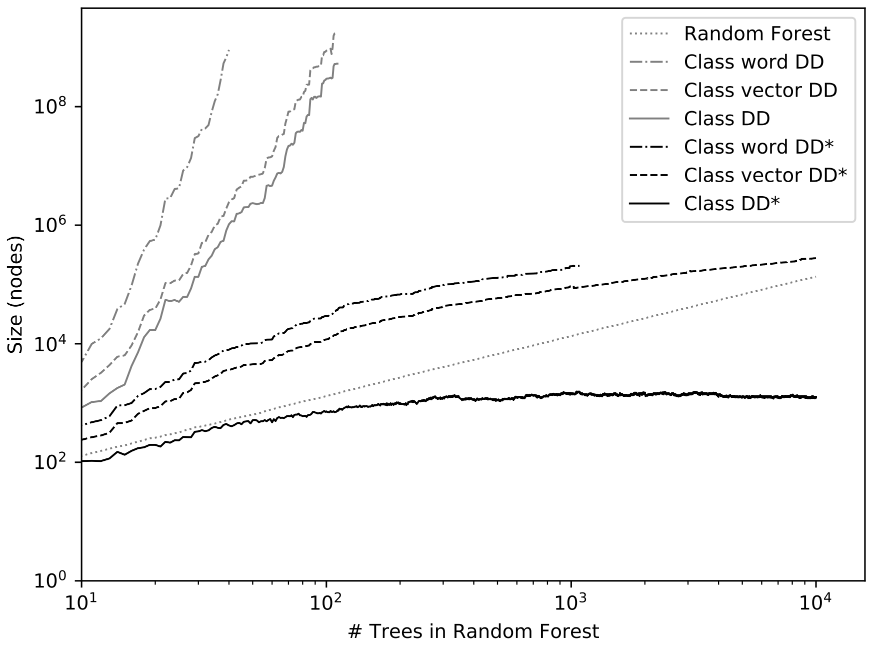


Figure 3.10: Sizes of the Random Forest and its semantically equivalent decision diagrams.

tion time significantly in comparison to the original Random Forest. This is due to the suppression of redundant predicate evaluations. In fact, the overall classification time is dominated by the linearly growing time to compute the most frequent class in each terminal word.

The reduction to just $|C|$ terminal nodes of the class vector-based variants has two effects:

- A partial collapse of the decision diagram: it is no longer essential which tree proposes which class, unifying all cases where the various classes are equally often proposed.
- Reduction to a constant overhead for the final aggregation step, in this case $|C|$.

The evaluation time reductions are again quite significant, only the space requirement got, like for the class word-based variant, out of hand very soon (Fig. 3.10), explaining the cut-off in Figure 3.9.

Whereas the previous two model structures can directly be computed compositionally, the most frequent label abstraction, i.e. the evaluation of the *majority vote* at compile time, can only be applied at the very end. Thus its construction has the same limitation as the class vector variant, and its impact on the size of the corresponding decision model is moderate (Fig. 3.10). Its impact on the evaluation time is, however, quite substantial (Fig. 3.9): Many of the internal decision nodes have become redundant by just focusing on the results of the majority vote.

Breathing semantics into the decision diagrams by unsatisfiable path elimination overcomes the scalability problems that are due to the enormous space requirements. In fact, it avoids the exponential blow-up in size in all three variants, with these diagrams even becoming significantly smaller than the original Random Forest (see DD* in Fig. 3.10). Moreover, classification times

100		
Dataset	**Random Forest**	**Final ADD**
Balance Scale	21,720	137 (-99.37%)
Breast Cancer	55,172	3,501 (-93.65%)
Lenses	1,518	11 (-99.28%)
Iris	1,312	722 (-44.97%)
Tic-Tac-Toe	55,232	1,563 (-97.17%)
Vote	9,768	1,337 (-86.31%)
1,000		
Dataset	**Random Forest**	**Final ADD**
Balance Scale	214,844	139 (-99.94%)
Breast Cancer	546,504	3,647 (-99.33%)
Lenses	14,132	11 (-99.92%)
Iris	13,492	1,458 (-89.19%)
Tic-Tac-Toe	570,976	1,593 (-99.72%)
Vote	97,770	1,168 (-98.81%)
10,000		
Dataset	**Random Forest**	**Final ADD**
Balance Scale	2,158,330	144 (-99.99%)
Breast Cancer	5,494,682	3,760 (-99.93%)
Lenses	136,986	11 (-99.99%)
Iris	135,952	1,267 (-99.07%)
Tic-Tac-Toe	5,670,532	1,529 (-99.97%)
Vote	988,358	1,148 (-99.88%)

Table 3.2: Decision diagram sizes for Random Forests of size 10.000 for other datasets [95].

are drastically reduced in all three cases (Fig. 3.1). In fact, the classification times eventually stabilise, illustrating the key feature of Random Forests, the reduction of the learner's variance (see DD* in Fig. 3.10). As sketched in Tables 3.1 and 3.2 these observations carry over to other popular data sets in the UCI Machine Learning Repository [95].

ADD-based program optimisation allows for quite substantial improvements in an application of practical relevance: Random Forests. Both, the reduction in running time and size of the final machine-learned model are impressive and exceed our initial expectations. This success was achieved on multiple popular datasets and proves the power of ADDs as an IR for program optimisation. Where the input format, in this case the Random Forest, facilitates a transformation to ADDs, the inherent optimisation and additional semantic optimisation unfold the true potential of the data structure. The hope is that this success is not limited to Random Forests but that it generalises further, to other implicit programs in the field of machine learning and beyond.

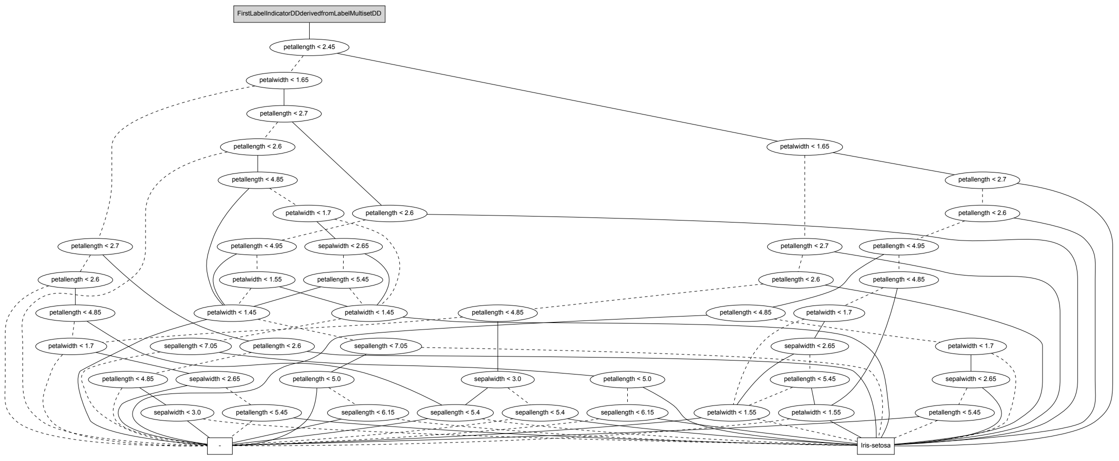


Figure 3.11: Explanation Binary Decision Diagram (BDD) for the class *Iris setosa* (15 nodes).

3.3 Towards Explainability

The initial motivation for the aggregation of Random Forests was to improve their running time and to reduce their size. However, the developed approach to aggregation and abstraction is also a powerful tool for another independent discipline: *Explainability* [2, 1]$_{\text{AP}}$. Unlike the original Random Forest, which is considered an uninterpretable black box model [29], the aggregated ADD condenses its semantics concisely. Like decision trees, the resulting tree-like structure can be considered an interpretable white box model [29]. Continuing the sequence of abstracting ADD transformations, we can generate even better explanations for the classification models, which focus on specific aspects.

Explainability is one of the hot topics in Machine Learning (ML) today. With increasingly complex machine-learned models like Deep Neural Networks (DNNs) and ensemble methods like Random Forests, it has become difficult to understand the decisions made by these models. At the same time, it is crucial to understand and explain exactly these decisions when they directly impact people's lives, may it be for credit standing, medical diagnosis, or automated driving.

With the ADD-based aggregation, we can achieve exactly this explainability goal for Random Forests and solve the following three problems [29]:

- The *Model Explanation Problem* is solved with the aggregated ADD that realises precisely the same classification function as the original Random Forest (see Fig. 3.8).
- The *Class Characterization Problem* is solved with a derived BDD that precisely characterises the forest's classification semantics with regard to one fixed class (See Fig. 3.11).
- The *Outcome Explanation Problem* is solved with a minimal conjunction of (negated) predicates that are sufficient to guide the sample into its result class according to the model.

The solution to the *model explanation* problem was essentially discussed in Section 3.2. The final aggregation is a single redundancy-free ADD that is as easy to understand as the usual decision trees. For this reason, it can be considered the most concise ADD and an ideal form of model explanation for random forests [29]. Figure 3.8 shows this model explanation for the running example of the Iris classification.

Continuing the sequence of abstracting transformations, we can also address a new notion of explainability, the *class characterisation* problem [1]$_{\text{AP}}$. Here, the model explanation is limited to just one of the classes, allowing for a simpler explanation of just one aspect. Class characterisation is based on a transformation of the model explanation model into a BDD that characterises only the chosen class, e.g. *Iris setosa*. Moving from the majority vote algebra to the standard Boolean logic via class projection naturally continues our line of abstraction. The resulting BDD is an explanation of the original Random Forest's behaviour that provides a precise and very focused explanation for when a certain class is chosen.

Figure 3.11 shows the class characterisation for the class *Iris setosa* which has only 15 nodes, in comparison to the 50 nodes of the model explanation (Fig. 3.8). In fact, the sum of the number of nodes in the class characterisations for all three classes is smaller than 50, which indicates the potential for a corresponding *semantic decomposition* of the model explanation.

Class characterisation is particularly interesting because it allows us to *reverse* the classification process: instead of determining a class for a certain sample, we obtain a characterisation of the set of all samples that will be classified as the given class. This change of perspective may have an important impact, e.g., in marketing contexts for switching from a customer to a product perspective [2].

For a responsible use of automatically derived classifications, the *Outcome Explanation Problem* is essential [29]. Class characterisations allow us to solve this problem in two further steps:

- **Path-based explanation.** Focusing on just one input at a time allows us to further refine the obtained explanation. When evaluating the aggregated BDD resulting from the first step, the conjunction of the components of the corresponding predicate path, i.e., of the (negated) predicates along the classification trace, provides a sufficient condition for the decision made. E.g., considering the sample

  $$petallength = 2.6,\ petalwidth = 1.5,\ petallength = 2.65,\ sepallength = 6.9,$$

  we obtain

  $$\begin{aligned} & petallength \geq 2.45 \\ \wedge\ & petalwidth \geq 1.45\ \wedge\ petalwidth < 1.65 \\ \wedge\ & petallength \geq 2.6\ \wedge\ petallength < 2.7 \\ \wedge\ & sepallength < 7.05. \end{aligned}$$

- **Simplifying conjunctions.** The collected predicates along a trace may yield redundancies – even when unsatisfiable paths were eliminated from the ADD. Removing choices from the conjunction as long as redundant choices exist yields a minimal explanation of the finally predicted class. In our example, $petallength \geq 2.45$ is redundant relative to the stricter predicate $petallength \geq 2.6$, which overall leads to a conjunction with five predicates.

3.4 General-Purpose Programming Languages

ADDs have already proven their potential as an IR for suitable program domains, exemplified by the discussed DSLs as well as the machine-learned model of a Random Forests. In both cases, the input program is of a very specific domain whose specific properties could be exploited for the transformation to ADDs and even for subsequent optimisations on that IR. These results are quite impressive, however, they are focused on the very specific program domains that they were developed for. The question arises if these approaches are applicable to more general program domains, and possibly to programs written in some general-purpose programming language.

With the shift from a specific domain to a general-purpose programming language, there are no longer any specific properties known about the program domain that naturally relate to ADDs as an IR in the compilation pipeline. Hence, we cannot exploit any specific properties of the input program but only those that come with the most general of all programming languages. The *while* language [96] is a common representative for such a general-purpose programming language and, for this reason, also serves as an input format here. Although the techniques in [3, 8, 6, 7]$_{\text{AP}}$ very directly rely on the relation between their input programs and the chosen IR, i.e. the ADDs, it is surprising how seamlessly some ideas generalise to the *while* language and consequently to many similar languages.

Traditionally, program optimisation techniques like redundancy elimination [18, 19, 20], code motion [21], strength reduction [97], constant propagation/folding [22], and loop invariant code motion [23] are quite syntax-oriented and mostly preserve program structure. Programs such as an iterative implementation of computing the n-th Fibonacci number typically remain untouched by such optimisations.

A transformation to ADDs as an IR, on the other hand, is quite different: It constitutes a new paradigm for program optimisation which is based on aggressive aggregation, i.e. on a partial evaluation-based decomposition of acyclic program fragments into a pair of computationally optimal structures: an ADDs to capture conditional branching and parallel assignments that refer to an Expression DAG (ED) which realises a redundancy-free computation.

The point of this decomposition is to obtain large program fragments which can be optimised using ADD technology, SMT solving, and expression normalisation without being *disturbed* by side effects. Not only are multiple occurrences of a term guaranteed to be semantically equivalent, like in Static Single Assignment (SSA) form [15], but the large size of the arising aggregated expressions further increases the optimisation potential.

This approach uses ADDs at its core and enables provably optimal transformations with some heuristic aspects in a transparent fashion:

1. First, cut points, similar to the ones in Floyd's inductive assertion method [98], are used to split the program into acyclic fragments. This step is heuristic and may be enhanced by, e.g., loop unrolling to increase the optimisation potential.

2. Second, the resulting acyclic fragments are symbolically executed, decomposing them into a *path condition* and a computational part in terms of a *parallel assignment*, a canonical procedure.

3. Finally, the path conditions are transformed into ADDs which are symbolically canonical

```
if ¬(1 < n) {
  fib := 1
} else {
  prev := 1;
  fib := 1;
  while 2 < n {
    tmp := prev + fib;
    prev := fib;
    fib := tmp;
    n := n - 1
  }
}
```

Figure 3.12: Iterative Fibonacci program in the *while* language (Def. 14)

for a fixed predicate order. The terminals of the ADDs hold the corresponding pairs of parallel assignment and successor cut point, allowing for program execution on the bases of this IR. Expressions that occur during this decomposition are stored in a dedicated data structure, the ED. Predicates of the ADD and right-hand sides of parallel assignments both simply reference nodes in this ED.

4. Optionally, further optimisations are applied such as a removal of infeasible paths in the ADD and a normalisation of the ED, both using an SMT solver, in our case Z3 [99]. This optimisation is similar to the elimination of infeasible paths in the transformed Random Forest ADDs.

Like all ADD-based program optimisations, we present also this one with the help of a running example: the well-known iterative implementation of the Fibonacci function (Fig. 3.12). This program is typically considered to lack any optimisation potential, yet with our ADD-based optimisation we can drastically improve its running time. The point here is that our technique supports loop unrolling as a first class optimisation technique: It is tailored to optimally aggregate large program fragments — especially those resulting from multiple loop unrollings — while only incurring the expected linear increase in size. In fact, we are able to achieve a performance improvement during the computation of the n-th Fibonacci number of more than an order of magnitude.

The running example is implemented in a simple *while* language [96] that we chose as a representative formalism for input programs.

Definition 14 (While Language) *Let V be a set of integer variables with $x \in V$. The* while *language comprises programs S according to the following Backus-Naur Form (BNF):*

$$\begin{aligned}
S &::= x := AE \mid \texttt{skip} \mid S;\ S \mid \texttt{if}\, BE\ \{S\}\ \texttt{else}\ \{S\} \mid \texttt{while}\, BE\ \{S\} \\
AE &::= V \mid \mathbb{Z} \mid AE + AE \mid AE - AE \mid AE * AE \mid AE/AE \\
BE &::= BE \vee BE \mid BE \wedge BE \mid \neg BE \mid AP \\
AP &::= AE < AE \mid AE = AE \mid true \mid false
\end{aligned}$$

We refer to program expressions of the form AE as arithmetic expressions, BE as Boolean expressions, and AP as atomic propositions.

In Figure 3.13, the Fibonacci program (Fig. 3.12) is equivalently presented in the form of a program graph [100]. Node *st* represents its start and node *te* its termination.

The key idea of our compilation approach is the radical decomposition of an input program into side-effect-free fragments that represent its control and data flow independently from one another. This decomposition involves three main steps:

- decomposition of the program graph into acyclic fragments using cut points along the lines proposed in Floyd [98],
- path-wise separation of the conditional control flow and the computational aspect through symbolic execution, and
- path aggregation to represent an entire acyclic program fragment in a single ADD.

For every acyclic fragment, these first three steps result in one ADD whose terminal nodes determine the next effect on the concrete program state and the point of continuation. An ED that comprises all the computations required anywhere in the fragment eliminates redundant arithmetic expressions. These two data structures together constitute the optimised program and allow for its rapid evaluation.

In the following, we show how to decompose a while program (Def. 14) into its control and data flow. In order to achieve this and also reason about the correctness of our decomposition, we introduce a Structural Operational Semantics (SOS) that combines concrete and symbolic domains and which processes a tuple $\langle S, c, c_H \sigma, \sigma_H \rangle$ containing

1. a while program S (Def. 14),
2. a Boolean variable c called *path-taken indicator* that states if the current path is the actually executed path w.r.t. the concrete program state,
3. a Boolean expression c_H (Def 14) called *path condition* that symbolically aggregates branching conditions on the current path,
4. a concrete program state σ that stores the current (integer) values of program variables, and
5. a symbolic program state σ_H called *parallel assignment* that stores variable values as arithmetic expressions based on the initial program state.

The path-taken indicator c (second component) and concrete program state σ (fourth component) are only added to our representation in order to facilitate reasoning on the correctness of our approach. Only the third and fifth components, namely the path condition and parallel assignment, are required for our decomposition-based compilation.

We distinguish concrete and symbolic program states along with their corresponding semantics as follows:

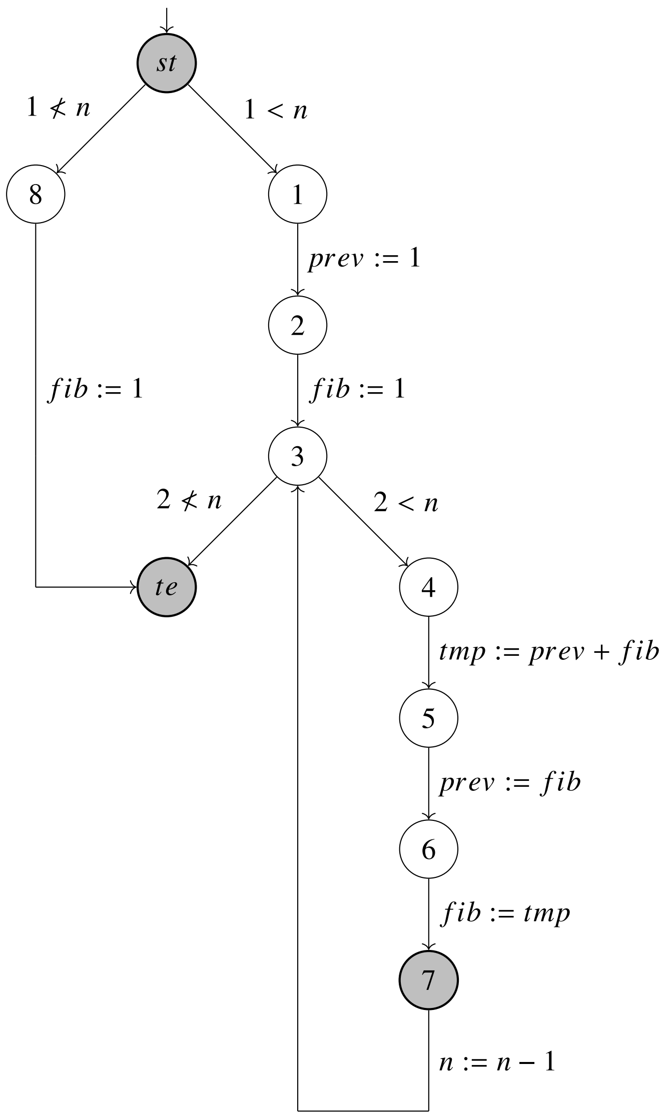


Figure 3.13: Program graph of our iterative Fibonacci program (Fig. 3.12).

Definition 15 (Concrete and Symbolic State) *Let V be a set of integer variables and AE denote arithmetic expressions (Def. 14) over symbolic versions of variables in V. Then a function $\sigma : V \longrightarrow \mathbb{Z}$ is called* concrete program state *whereas a function $\sigma_H : V \longrightarrow AE$ is called* symbolic program state.

For any expression $t \in (AE \cup BE)$, $[\![\ t \ ]\!](\sigma)$ denotes the evaluation of expression t with respect to the concrete program state σ, and $[\![\ t \ ]\!](\sigma_H)$ denotes the Herbrand interpretation [101, 102] of expression t with respect to the symbolic state σ_H.

The semantic evaluation $[\![\ \cdot \ ]\!]$ extends naturally to symbolic program states because the latter map variables to expressions. For any $x \in V$, the resulting function is defined as

$$(\ [\![\ \sigma_H \ ]\!](\sigma) \ )(x) := [\![\ \sigma_H(x) \ ]\!](\sigma).$$

The difference between symbolic and concrete state is best illustrated with the help of an example. Let x and y be two variables and let $t = x + y$ be an arithmetic expression. For a concrete program state $\sigma = \{ \ x \mapsto 5, y \mapsto 4 \ \}$, the semantics of t yield a numeric value $[\![\ t \ ]\!](\sigma) = 9$. With a symbolic program state $\sigma_H = \{ \ x \mapsto 3+2, y \mapsto X \ \}$, on the other hand, the very same term evaluates to a symbolic term $[\![\ t \ ]\!](\sigma_H) = (3+2) + X$ that is not further evaluated (Herbrand interpretation).

Note that the notions of symbolic execution [103] and iterated Herbrand interpretation are strongly related. We choose the latter for presentation due to its clear formal roots. Other than symbolic execution, our main approach does not partially evaluate expressions, for example the sum of two integer constants. However, such optimisations can of course be incorporated into our compilation.

For the two domains of concrete and symbolic states, respectively, we can define an SOS that executes a while program concretely and symbolically at the same time:

Definition 16 (Combined-Domain SOS) *For any (concrete or symbolic) program state σ, let $\sigma\{y/x\}$ denote the substitution of x by y in σ. Let '$\bullet$' above a Boolean operator denote that this* *operation is evaluated semantically. Then our Combined-Domain SOS is defined as follows:*

$$\textit{assign}\ \frac{-}{\langle x := t, c, c_H, \sigma, \sigma_H\rangle \longrightarrow \langle c, c_H, \sigma\{[\![\ t\ ]\!](\sigma)/x\}, \sigma_H\{[\![\ t\ ]\!](\sigma_H)/x\}\rangle}$$

$$\textit{skip}\ \frac{-}{\langle \texttt{skip}, c, c_H, \sigma, \sigma_H\rangle \longrightarrow \langle c, c_H, \sigma, \sigma_H\rangle}$$

$$\textit{comp}_1\ \frac{\langle S_1, c, c_H, \sigma, \sigma_H\rangle \longrightarrow \langle S_1', c', c_H', \sigma', \sigma_H'\rangle}{\langle S_1;\ S_2, c, c_H, \sigma, \sigma_H\rangle \longrightarrow \langle S_1';\ S_2, c', c_H', \sigma', \sigma_H'\rangle}$$

$$\textit{comp}_2\ \frac{\langle S_1, c, c_H, \sigma, \sigma_H\rangle \longrightarrow \langle c', c_H', \sigma', \sigma_H'\rangle}{\langle S_1;\ S_2, c, c_H, \sigma, \sigma_H\rangle \longrightarrow \langle S_2, c', c_H', \sigma', \sigma_H'\rangle}$$

$$\textit{if}_1\ \frac{-}{\langle \texttt{if}\, b\ \{S_1\}\ \texttt{else}\ \{S_2\}, c, c_H, \sigma, \sigma_H\rangle \longrightarrow \langle S_1, c \overset{\bullet}{\wedge} [\![\ b\ ]\!](\sigma), c_H \wedge [\![\ b\ ]\!](\sigma_H), \sigma, \sigma_H\rangle}$$

$$\textit{if}_2\ \frac{-}{\langle \texttt{if}\, b\ \{S_1\}\ \texttt{else}\ \{S_2\}, c, c_H, \sigma, \sigma_H\rangle \longrightarrow \langle S_2, c \overset{\bullet}{\wedge} \overset{\bullet}{\neg}[\![\ b\ ]\!](\sigma), c_H \wedge \neg[\![\ b\ ]\!](\sigma_H), \sigma, \sigma_H\rangle}$$

$$\textit{while}\ \frac{-}{\langle \texttt{while}\, b\ \{S\}, c, c_H, \sigma, \sigma_H\rangle \longrightarrow \langle \texttt{if}\, b\ \{S;\ \texttt{while}\, b\ \{S\}\}\ \texttt{else}\ \{\texttt{skip}\}, c, c_H, \sigma, \sigma_H\rangle}$$

As usual, the notation $\langle S, c, c_H, \sigma, \sigma_H\rangle \longrightarrow^* \langle S', c', c_H', \sigma', \sigma_H'\rangle$ *denotes a sequence of SOS rule applications.*

Figure 3.14 illustrates the application of our SOS rules based on the fragment

$$n := n - 1;\ \texttt{while}\ 2 < n\ \{tmp := prev + fib;\ prev := fib;\ fib := tmp;\ n := n - 1\}$$

of our Fibonacci program up to the program's termination (left branch) or a repetition of that fragment (right branch). Only the right branch would be taken based on the concrete program state. This fragment therefore represents the paths from node 7 to nodes te and 7 in the program graph of Figure 3.13. Corresponding program fragments (first component of the SOS tuple) are omitted in favour of edge annotations. Upper case letters represent initial symbolic values of program variables, e.g. F is the initial symbolic value of the program's variable fib. The difference between concrete and symbolic semantics becomes apparent when comparing the program states σ and σ_H. The Combined-Domain SOS (Def. 16) is equivalent to the common concrete SOS [104] when (i) ignoring the components c, c_H, and σ_H as well as (ii) adding the side condition $[\![\ b\ ]\!](\sigma) = tt$ to rule if$_1$ and $[\![\ b\ ]\!](\sigma) = \mathit{ff}$ to rule if$_2$. Note that any path of the Combined-Domain SOS would be part of the common concrete SOS if and only if $c = tt$.

Based on these SOS rules, we can formulate the main correctness theorem of this decomposition:

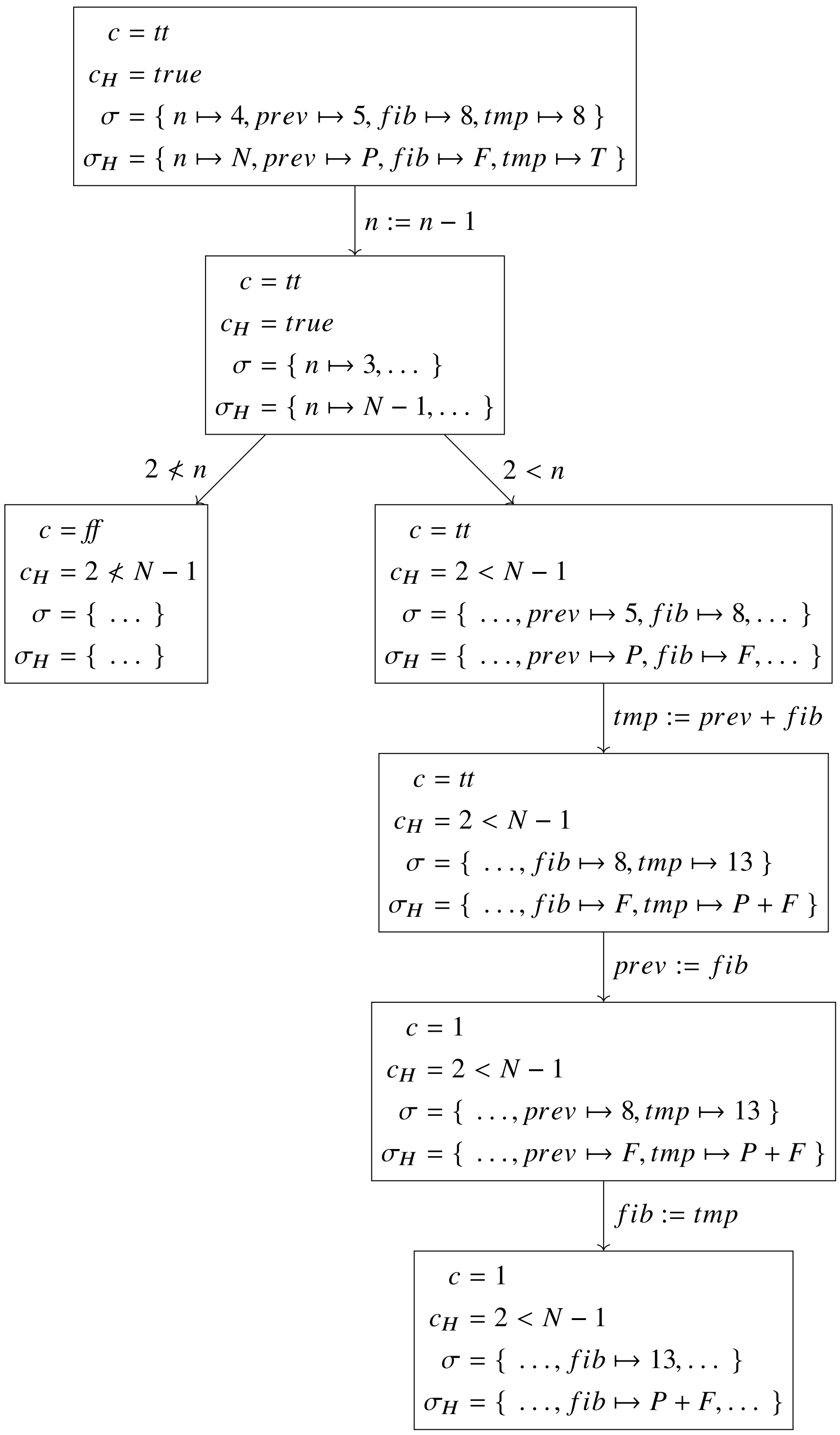

$c = tt$
$c_H = true$
$\sigma = \{ n \mapsto 4, prev \mapsto 5, fib \mapsto 8, tmp \mapsto 8 \}$
$\sigma_H = \{ n \mapsto N, prev \mapsto P, fib \mapsto F, tmp \mapsto T \}$
$n := n - 1$
$c = tt$
$c_H = true$
$\sigma = \{ n \mapsto 3, \dots \}$
$\sigma_H = \{ n \mapsto N - 1, \dots \}$
$2 \nless n$
$2 < n$
$c = ff$
$c_H = 2 \nless N - 1$
$\sigma = \{ \dots \}$
$\sigma_H = \{ \dots \}$
$c = tt$
$c_H = 2 < N - 1$
$\sigma = \{ \dots, prev \mapsto 5, fib \mapsto 8, \dots \}$
$\sigma_H = \{ \dots, prev \mapsto P, fib \mapsto F, \dots \}$
$tmp := prev + fib$
$c = tt$
$c_H = 2 < N - 1$
$\sigma = \{ \dots, fib \mapsto 8, tmp \mapsto 13 \}$
$\sigma_H = \{ \dots, fib \mapsto F, tmp \mapsto P + F \}$
$prev := fib$
$c = 1$
$c_H = 2 < N - 1$
$\sigma = \{ \dots, prev \mapsto 8, tmp \mapsto 13 \}$
$\sigma_H = \{ \dots, prev \mapsto F, tmp \mapsto P + F \}$
$fib := tmp$
$c = 1$
$c_H = 2 < N - 1$
$\sigma = \{ \dots, fib \mapsto 13, \dots \}$
$\sigma_H = \{ \dots, fib \mapsto P + F, \dots \}$

Figure 3.14: Execution of SOS rules (Def. 16) based on a loop program fragment of the iterative Fibinacci program (Fig. 3.12). This fragment extends from node 7 to nodes *te* (left branch) and 7 (right branch) in Figure 3.13.

Theorem 4 (Correctness of Control-Data-Decomposition) *Let id denote the symbolic program state that maps each $x \in V$ to its symbolic version. Let*

$$\langle S, c, c_H, \sigma, id\rangle \longrightarrow^* \langle S', c', c'_H, \sigma', \sigma'_H\rangle$$

with $c = [\![c_H]\!](\sigma)$. Then the following holds for all expressions $t \in (AE \cup BE)$:

$$[\![t]\!](\sigma') = [\![[\![t]\!](\sigma'_H)]\!](\sigma)$$

Intuitively, Theorem 4 states that the evaluation of an expression t interpreted via σ_H with respect to the initial state σ is semantically equivalent to evaluating t in the current state σ'[2]. The theorem holds regardless of whether or not a given chain of SOS rule applications is part of the traditional concrete SOS semantics. A corresponding proof follows from a straightforward induction over the SOS rules (Def. 16) and the corresponding expressions (Def. 14).

The following corollary follows directly from Theorem 4 and the fact that our path-taken indicator c could alternatively be stored as a Boolean variable in the program state:

Corollary 1 *Let V be the set of program variables and let id denote the symbolic program state that maps each $x \in V$ to its symbolic version. Let tt denote the semantic value of Boolean constant $true$ and let*

$$\langle S, tt, true, \sigma, id\rangle \longrightarrow^* \langle S', c', c'_H, \sigma', \sigma'_H\rangle.$$

Then, the following holds:

$$\begin{aligned} c' &= [\![c'_H]\!](\sigma) \\ \textit{and}\ \ \sigma' &= [\![\sigma'_H]\!](\sigma). \end{aligned}$$

The first part of this corollary asserts that our path condition is correct, the second part is the reason why we call the symbolic program state *parallel assignment*: Each variable can be updated independently of others because for each such $x \in V$, the expression $\sigma'_H(x)$ is only based on symbolic versions of variables in V and constants. This side-effect-free update mechanism together with our aggregated path condition yields a decomposition of our program into data and control flow, respectively.

In the end, we want to apply the discussed decomposition to entire programs of the *while* language. The goal is a static transformation of a given program into a version where control and data are fully decomposed. Alone with the SOS rules (Def. 16), this is not possible: We do not have a bound for the length of an execution path and a static decomposition would therefore not terminate.

In order to circumvent this problem, we segment a given input program into acyclic fragments which we then decompose individually. Our method is similar to the path variant of Floyd's inductive assertion method [98] for program verification: We choose as cut points

- the start node,

[2]This is an instance of a well-known substitution lemma.

- the termination node, and
- one node for each while construct

of a given while program (Def. 14) to ensure that they interrupt all loops. Based on such a choice, every cut point has a statically known number of outgoing paths to successor cut points, and these outgoing inter-cut point paths have a statically known length. Because of these static bounds, a purely symbolic execution of such an acyclic fragment starting at a cut point u is guaranteed to terminate. To execute a program fragment S symbolically means to execute the SOS rules (Def. 16) while (i) ignoring the path-taken indicator c and concrete program state σ, and (ii) starting with $\langle S, \cdot, true, \cdot, id \rangle$.

We can use these acyclic program fragments between cut points to execute the original program while preserving its semantics:

Theorem 5 (Compositionality) *Let*

$$\langle S, tt, true, \sigma, id \rangle \longrightarrow^* \langle S', tt, c'_H, \sigma', \sigma'_H \rangle,$$
$$\langle S', tt, true, \sigma', id \rangle \longrightarrow^* \langle S'', tt, c''_H, \sigma'', \sigma''_H \rangle.$$

Then the following holds:

$$\sigma'' = [\![\sigma''_H]\!]([\![\sigma'_H]\!](\sigma)).$$

This theorem follows straightforwardly from two applications of Corollary 1.

In a while program, every node in the corresponding program graph [100] can be annotated with a fragment of that program. [3] As a consequence, we can specify cut points visually as nodes in that graph.

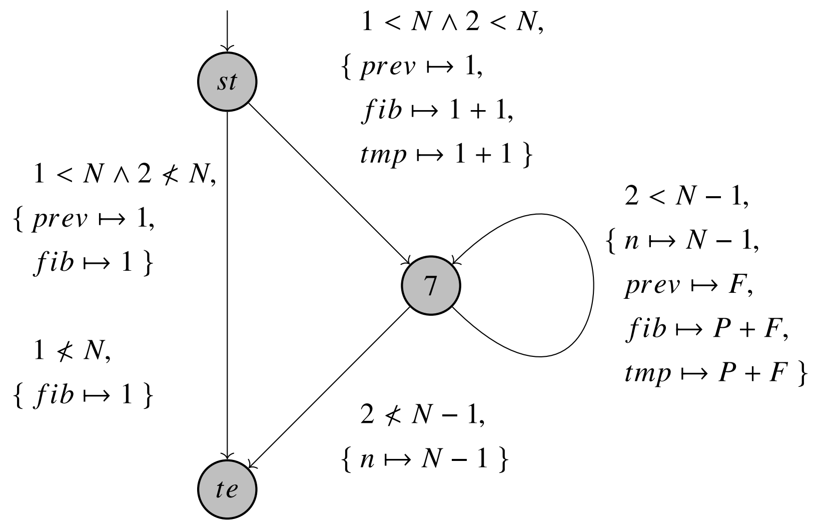
Figure 3.13 illustrates a choice of cut points for our Fibonacci program as nodes that are coloured grey. In addition to start node st and termination node te, we choose node 7 as a cut point in order to interrupt the only while loop $3 \to 4 \to 5 \to 6 \to 7 \to 3$ in this program. The program fragment corresponding to cut point 7 and the matching SOS rule application up to successor cut points are the same as used in Figure 3.14.

The actual choice of cut points for a given input program is fundamental for later steps in our optimisation process. This set of cut points therefore serves as a parameter of our optimisation relative to which we can achieve optimal results, but which we choose heuristically. Usually, the longer the fragments of a given program based on chosen cut points are, the more potential for optimisation exists.

Note that in our Fibonacci program, choosing node 7 (instead of 3) to interrupt its loop results in the path $st \to 1 \to 2 \to 3 \to te$ to be kept uninterrupted by cut points. This means that our approach allows us to bypass a loop's cut point if that loop's body will not be entered.

With the segmentation of the original program into acyclic fragments and their subsequent piecewise symbolic execution, we have obtained a finite number of program fragments, each characterised by the symbolic execution traces up to their successor cut points:

[3]Note that due to the SOS rule for the while construct, a fragment of a program does not necessarily have to be a substring of that program.



st
7
te
$1 < N \wedge 2 < N$, { $prev \mapsto 1$, $fib \mapsto 1+1$, $tmp \mapsto 1+1$ }
$1 < N \wedge 2 \not< N$, { $prev \mapsto 1$, $fib \mapsto 1$ }
$1 \not< N$, { $fib \mapsto 1$ }
$2 < N-1$, { $n \mapsto N-1$, $prev \mapsto F$, $fib \mapsto P+F$, $tmp \mapsto P+F$ }
$2 \not< N-1$, { $n \mapsto N-1$ }

Figure 3.15: Contracted cut point paths for our Fibonacci program (Figure 3.12).

Definition 17 (Contracted Cut Point Path) *Let u be a cut point associated with the program fragment S and let u' be one of its successor cut points associated with the program fragment S'. Moreover, let*

$$\langle S, \cdot, true, \cdot, id \rangle \longrightarrow^* \langle S', \cdot, c_H, \cdot, \sigma_H \rangle.$$

We call

$$u \overset{c_H, \sigma_H}{\longrightarrow} u'$$

a (contracted cut point) path from u to u' with the path condition c_H and the parallel assignment σ_H.

Figure 3.15 illustrates this contracted view on our running example, the Fibonacci program (Fig. 3.12). For brevity, we omit all variables in the parallel assignments that remain unaffected, i.e. those that map variables to their respective symbolic version.

Because contracted cut point paths are free of side effects, the compositional aggregation of a cut point's outgoing paths yields a very natural fragment-wise transformation of the input program:

- We will first transform these paths individually to ADDs.
- On this basis, we are able to aggregate them into a single ADD per cut point that completely defines the corresponding program fragment's behaviour. This aggregation is very similar to the aggregation of a Random Forest's trees into a single aggregated ADD (Sec. 3.2).

Just like we considered one tree after the other in the Random Forest transformation, we will here consider one contracted cut point path after the other. Let us first consider one such path

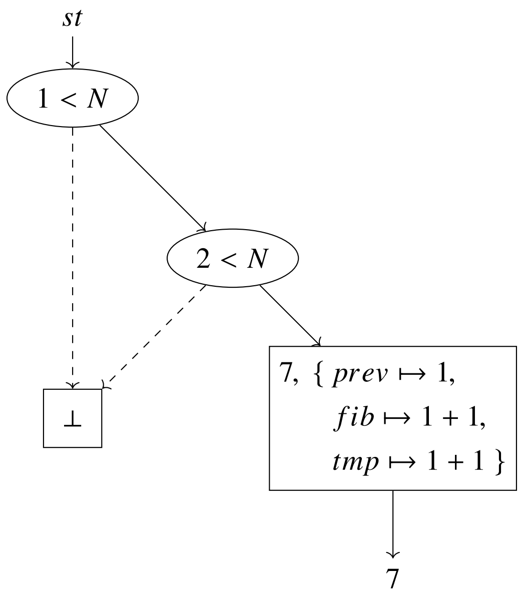


Figure 3.16: Exemplary ADD for the contracted cut point path from node *st* to node 7 in Figure 3.15 with the path condition $1 < N \land 2 < N$.

and its respective path condition. The Boolean expression over atomic propositions (Def. 14) can be represented by means of a BDD.

Because the atomic propositions that appear in our path conditions are not necessarily independent from one another, canonicity and minimality are only guaranteed on the level of their Herbrand interpretation. Despite this, decision diagrams have proven to be an effective representation, already in the Random Forest transformation.

Figure 3.16 visualises the decision diagram that results from the example path condition $1 < N \land 2 < N$ (Fig. 3.15). Note that this diagram is not exactly a BDD as its terminal nodes differ from Boolean values. Instead, we use ADDs for two reasons:

- For every path, we intend to store its effect on the program state, i.e. (i) the parallel assignment and (ii) the subsequent cut point, directly in the terminal nodes.
- We associate these decision diagrams with an algebraic structure that facilitates their composition. This is in fact key to our aggregation of a cut point's outgoing paths.

Both, parallel assignment and successor cut point, are applicable in case the path condition holds and they are irrelevant otherwise. Hence, it is only natural to substitute them for the BDD's 1-terminal and, for distinguishability, we also replace the BDD's 0-terminal with a dedicated $\perp$ element. In this way, any contracted path can be transformed to an ADD and we denote this transformation with dd. The result is an ADD that is structurally analogous to the path condition BDD. The underlying algebraic structure is, in this case, a lattice:

Definition 18 (Path Aggregation Lattice) *Let U be a set of cut points and let Σ_H be a set of all parallel assignments. We define the path aggregation lattice $\mathcal{A} = ((U \times \Sigma_H) \cup \{ \bot, \top \}, \circ)$ with its supremum*

$$a \circ b := \begin{cases} a & \textit{if } a = b \neq \bot \\ a & \textit{if } b = \bot \neq a \\ b & \textit{if } a = \bot \neq b \\ \top & \textit{otherwise.} \end{cases}$$

$\mathcal{A}$ forms a flat lattice with $\bot$ and $\top$ as its least and greatest element, respectively. Note that $\top$ serves no purpose here other than the natural completion of the structure and will never appear in our aggregation process. The least element $\bot$ on the other hand does appear, but only in intermediate results. Intuitively, we understand $\bot$ as the *undefined* case in which the path condition does not hold.

The transformations of path conditions to BDDs and finally to ADDs constitutes a change in granularity. Where predicates were previously expressed by means of possibly complex Boolean expressions (BE in Def. 14), they are now based on atomic propositions only (AP in Def. 14). Any complexity of Boolean formulas beyond its APs is delegated to well-studied and efficient ADD routines in a service-oriented fashion.

To allow for a simultaneous evaluation of path conditions, we aggregate all outgoing paths per cut point. With the supremum operation $\circ$ we can achieve this in a very simple way: We collect the most concrete information among all paths. If either of them is undefined, i.e. $\bot$, we adopt the other, more concrete definition.

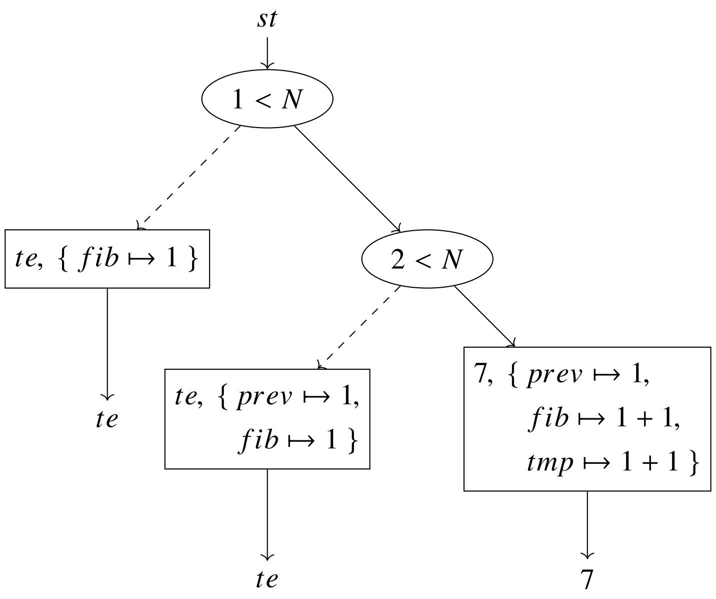
Being defined on their co-domain, we can easily apply $\circ$ to the previously constructed path ADDs. For a given cut point u with its outgoing paths $p_0, p_1, \ldots, p_{n-1}$, we can construct the aggregated path ADD for u as the repeated application of $\circ$:

$$dd(u) := dd(p_0) \circ dd(p_1) \circ \cdots \circ dd(p_{n-1}).$$

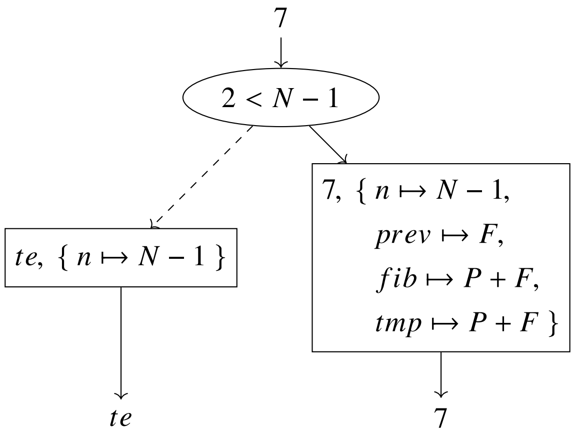
Figures 3.17a and 3.17b show the final ADDs per cut point for our running example. Every non-termination cut point yields one ADD with exactly one terminal node for each of its successor cut points. Note that these ADDs are not necessarily trees but DAGs that share common substructures to keep the data structure small — an effect that becomes important for more complex path conditions.

Because the path conditions induce a partitioning on the Herbrand interpretation of atomic propositions, the $\top$ element will never appear in the aggregation process. For the same reason, all undefined cases $\bot$ will eventually dissolve and the resulting ADDs yield only concrete pairs of parallel assignment and successor cut point.

With ADDs, we have found a program representation that allows us to simultaneously evaluate all relevant path conditions. At the same time, the program's semantics remain untouched. Let $[\![\cdot]\!]_A(\sigma)$ denote the standard semantic function to evaluate an ADD that, in our case, yields a parallel assignment based on some concrete program state σ (Def. 15). The correctness of the aggregated path ADDs follows directly from Theorem 5:


st
1 < N
te, { fib ↦ 1 }
te
2 < N
te, { prev ↦ 1, fib ↦ 1 }
te
7, { prev ↦ 1, fib ↦ 1 + 1, tmp ↦ 1 + 1 }
7

(a) ADD for the start cut point *st*.



(b) ADD for the inner cut point 7.

Figure 3.17: Aggregated path ADDs of the Fibonacci program.

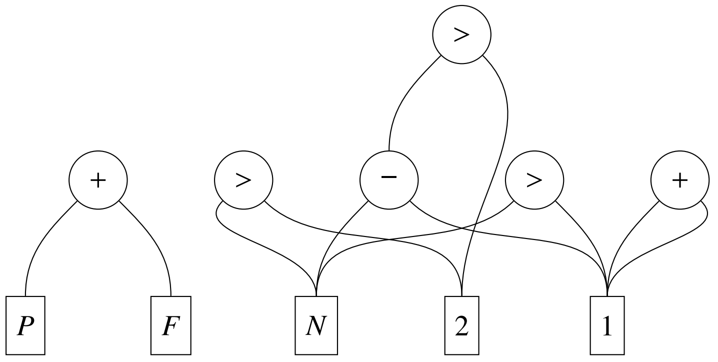


Figure 3.18: Expression DAG (ED) for our Fibonacci program (Figure 3.12).

Theorem 6 (Correctness of Path Aggregation) *Let u be some cut point associated with the program fragment S, let $p_0, p_1, \ldots, p_{n-1}$ denote its outgoing paths, and let*

$$\langle S, tt, true, \sigma, id\rangle \longrightarrow^{*} \langle S', tt, c'_H, \sigma', \sigma'_H\rangle.$$

Then the following holds:

$$\sigma' = [\![\, [\![\, dd(p_0) \circ dd(p_1) \circ \cdots \circ dd(p_{n-1}) \,]\!]_A(\sigma) \,]\!](\sigma).$$

Intuitively, Theorem 6 justifies the use of the aggregated ADD representation to evaluate all relevant path conditions simultaneously. Its proof is straightforward but tedious by induction over the aggregated paths and the ADD structure.

The aggregation of paths into equivalent ADDs resolves redundancies among the path conditions per cut point. The problem of duplicate arithmetic expressions, however, remains. These appear not only in the parallel assignments of the ADDs' terminals but also in their inner nodes' predicates, i.e. in the atomic propositions. In fact, the duplication of arithmetic terms is a typical outcome of symbolic substitution, an operation that we heavily rely on during the course of our partial evaluation. It is therefore crucial to also eliminate these redundancies.

Achieving this goal is simple: We resolve duplications in the form of an Expression DAG (ED). Every constant and every program variable becomes a unique node — the atoms of this data structure. Based on these, the remaining expressions can be represented uniquely and with references to their respective sub-expressions.

Figure 3.18 shows the ED for our Fibonacci program (Fig. 3.12). With additional optimisations, especially loop unrolling and expression normalisation, the number of expressions may grow quite drastically.

With ADDs and the ED, we have found an aggregated representation that finally reaches our goal: the efficient execution of a given program.

For any given cut point u and concrete program state σ, we can execute the optimised representation of the corresponding program fragment: We start at the root of u's aggregated path ADD and evaluate its atomic propositions based on the concrete program state σ. The effect of the program fragment is not applied until a terminal is reached. Only then, the parallel assignment is applied to the program state σ. This execution is semantically equivalent to that of the original program fragment (Theorem 6).

This fragment-wise execution is sufficient to also execute the original program in its entirety. Starting with some program state σ at the initial cut point st, we effectively jump from cut point to cut point until the termination node te is reached (Fig. 3.15). This iterative piecewise execution is then semantically equivalent to that of the entire input program (Theorem 5).

The program decomposition introduces a radically new compilation paradigm. It is therefore not surprising that this method benefits from different optimisation techniques than classical compilers do.

As we optimise the program on the bases of acyclic program fragments, loop unrolling becomes a boosting factor to the success of this program optimisation. The resulting larger acyclic fragments facilitate the success of subsequent optimisation. Its effect is the possibility for direct jumps over multiple loop iterations in the optimised program representation. This technique and its effect in this context is discussed in more detail in [4]$_{\mathrm{AP}}$.

Similarly to the Random Forest transformation [3]$_{\mathrm{AP}}$, also here, infeasible path elimination has the potential to greatly simplify our aggregated program representation, the ADDs. Loop unrolling and predicate reordering, in particular, yield infeasible paths and subsequent treatment can greatly reduce the diagrams' depths and sizes. In general, their origin is threefold:

- Already the input program may contain infeasible paths between its cut points, or even dead code.
- As a result of loop unrolling, longer paths are considered, some of which may be infeasible.
- Enforcing a particular predicate order may swap atomic propositions which, again, may result in infeasible paths — this time, however, only the aggregated path ADDs are affected.

It is desirable to detect infeasibilities early in the optimisation pipeline. Already at the stage of symbolic execution, we can validate path conditions for their satisfiability using an SMT solver. Rather than fully expanding the execution tree, we discard irrelevant paths early and effectively cut off all their continuations. This speeds up not only the optimisation process, but, more importantly, yields smaller ADDs that are faster to evaluate.

The repeated symbolic substitution, especially through multiple iterations of a loop, generates a great amount of unique arithmetic expressions. Moving from a purely syntactic view to a more semantic understanding, we can exploit their inherent potential for simplification and condense these terms to a close to normal form. We delegate this generally non-trivial task to established SMT solving techniques.

As a result, the overall number of arithmetic operations is reduced. This normalisation also evaluates expressions already at compile time where possible. At its core, SMT technology essentially exploits common arithmetic laws and, in case of the Fibonacci example, typically transforms expressions to their equivalent polynomials:

$$(((n-1)-1)-1) = n-3$$
$$(P+F)+(F+(P+F)) = 3F+2P.$$

At the same time, this simplification serves a second purpose: Semantically equivalent expressions become unified, also syntactically. This effect cascades to atomic propositions and reduces the overall number of syntactically different predicates in the ADDs where expression normalisation allows for a smaller and shallower representation. EDs are affected in a similar fashion: Where the normalisation unifies terms, it also merges nodes in the EDs.

We have discussed four additional optimisations and their anticipated interplay with one another. Figure 3.19a shows the effect of loop unrolling, predicate reordering, and expression normalisation on our Fibonacci program. In this example, the loop was unrolled twice and the predicate order was chosen at random. Arithmetic expressions were condensed to simple forms such as $N - 3$ or $3F + 2P$, both of which were originally nested.

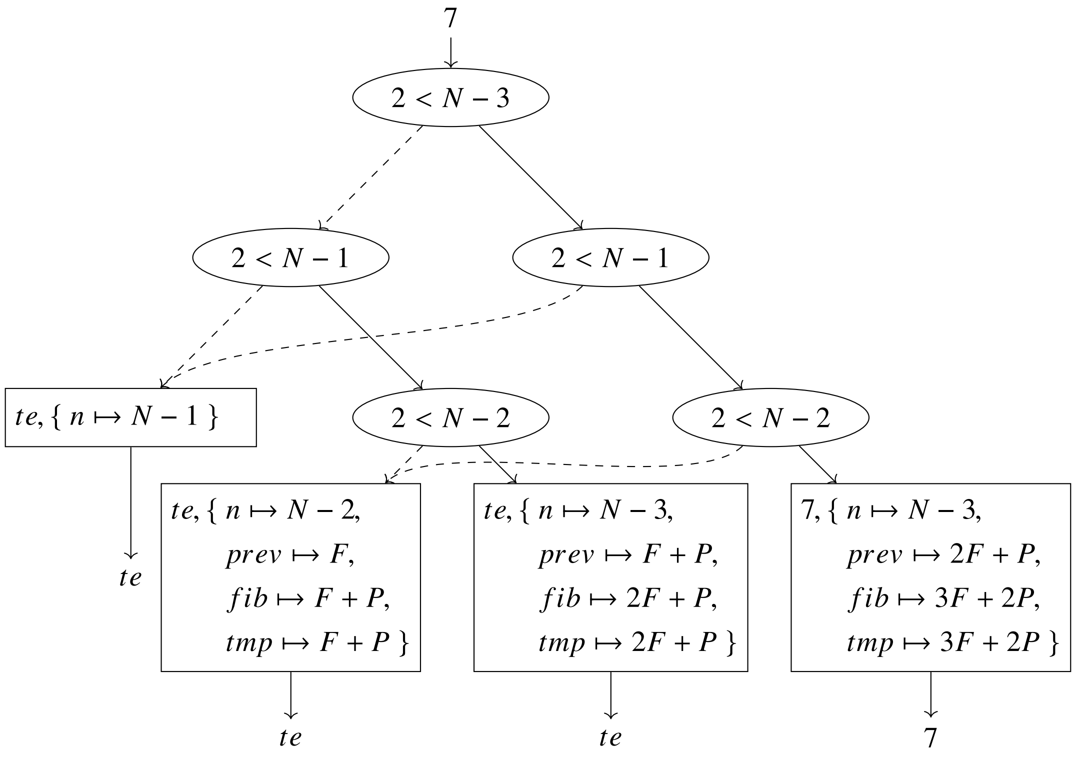
The resulting ADD contains two infeasible paths that, in this case, result solely from the loop unrolling. When $2 < N - 3$ holds, the truth value of the following two predicates $2 < N - 1$ and $2 < N - 2$ is already determined. Paths like these unnecessarily complicate the ADD and are, for this reason, subsequently eliminated. The result is shown in Figure 3.19b: a smaller and shallower equivalent. An important effect of the presented predicate order is that the biggest step through the unrolled loop can be evaluated with only one atomic proposition: $2 < N - 3$.

To give an impression of the capabilities of this ADD-based program optimisation technique, we chose the iterative implementation of the Fibonacci function (Fig. 3.12) that appears to lack any potential for optimisation. We evaluate our aggregated ADD representation under consideration of the additional optimisations — loop unrolling in particular. Our goal here is a machine-independent evaluation of the running time to not rely on implementation details at this stage. We achieve this with a simulated infinite register machine where we account cost of one per instruction:

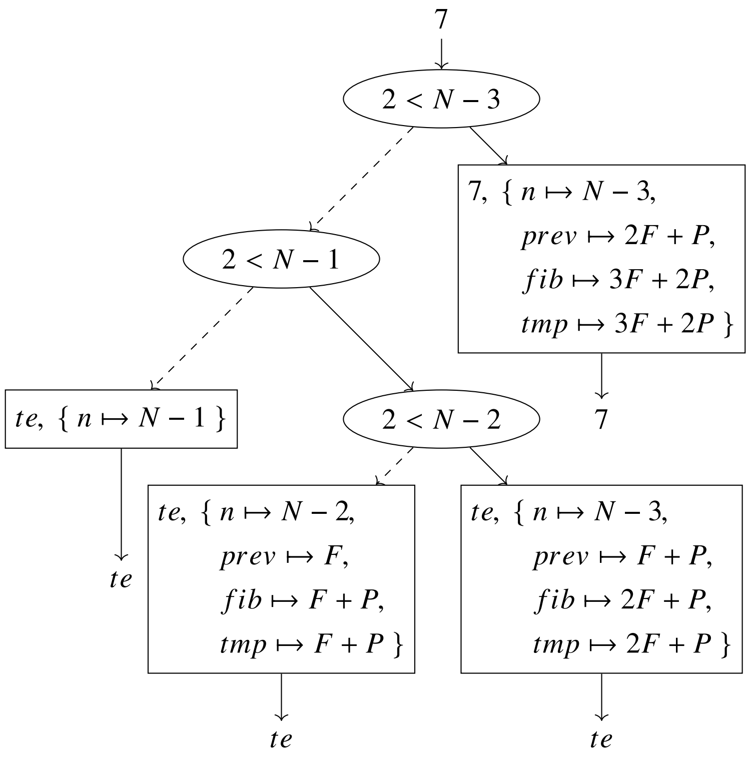
- arithmetic and logic operations,
- conditional jump, and
- assignment.

We first examine the execution time measured for the original program in comparison to optimised versions produced by our compiler in which the program's loop was unrolled up to 4, 16, or 64 times, respectively. In cases in which our predicate reordering that involves randomisation is applied, we report the average of 1000 unique measurements. Figure 3.20 shows the execution times for the original program and the aggregated versions that were optimised by our compiler with all of the discussed optimisations: expression normalisation, infeasible path elimination, and our predicate reordering.

Already in the case of 4 loop unrollings, we can observe a declining execution time in comparison to the original program. Figure. 3.20 clearly shows that variants in which a higher amount of loop unrolling was performed entail a larger speedup if n exceeds the number of unrollings. While for small n, loop unrolling might incur minor execution time overhead due to an increased average path length from the ADD's root to its terminals, this overhead is easily compensated by a drastic speedup for larger n. For the case of 64 loop unrollings and the computation of $fib(150)$, the measured execution time can be reduced by a factor of 13 in comparison to the original program.


7
2 < N − 3
2 < N − 1
2 < N − 1
te, { n ↦ N − 1 }
2 < N − 2
2 < N − 2
te
te, { n ↦ N − 2, prev ↦ F, fib ↦ F + P, tmp ↦ F + P }
te, { n ↦ N − 3, prev ↦ F + P, fib ↦ 2F + P, tmp ↦ 2F + P }
7, { n ↦ N − 3, prev ↦ 2F + P, fib ↦ 3F + 2P, tmp ↦ 3F + 2P }
te
te
7

(a) With 2 loop unrollings, optimised predicate order, and expression normalisation.


7
2 < N − 3
7, { n ↦ N − 3, prev ↦ 2F + P, fib ↦ 3F + 2P, tmp ↦ 3F + 2P }
2 < N − 1
7
te, { n ↦ N − 1 }
2 < N − 2
te
te, { n ↦ N − 2, prev ↦ F, fib ↦ F + P, tmp ↦ F + P }
te, { n ↦ N − 3, prev ↦ F + P, fib ↦ 2F + P, tmp ↦ 2F + P }
te
te

(b) With 2 loop unrollings, optimised predicate order, infeasible path elimination, and expression normalisation.

Figure 3.19: Impact of additional optimisations on the aggregated path ADD.

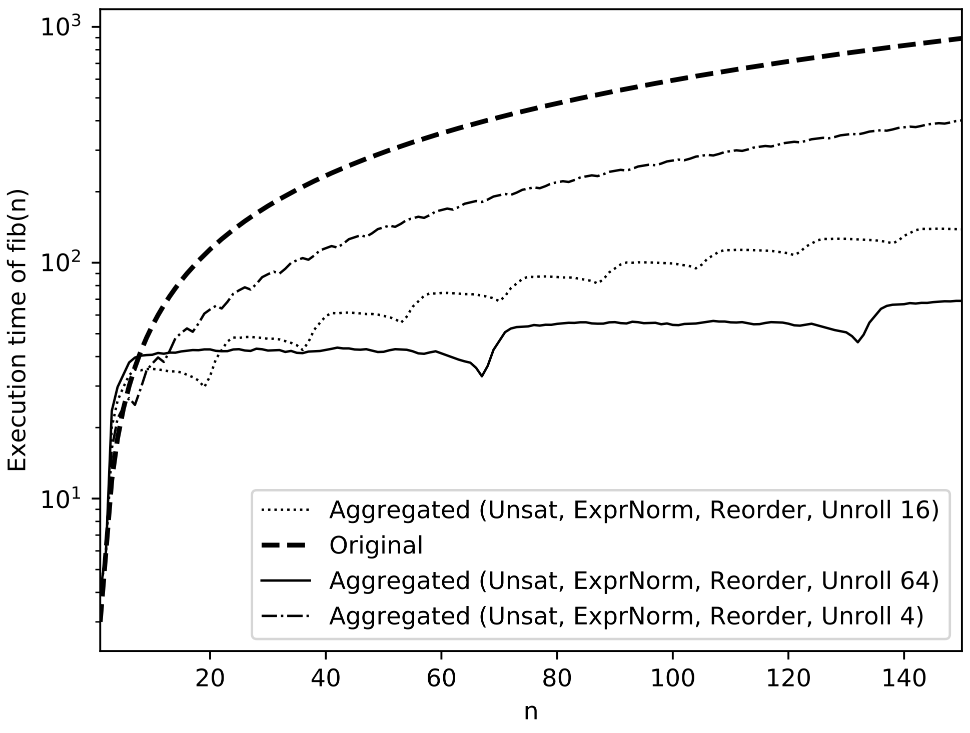


Figure 3.20: Execution times of versions with different numbers of loop unrolling while enabling all optimisations.

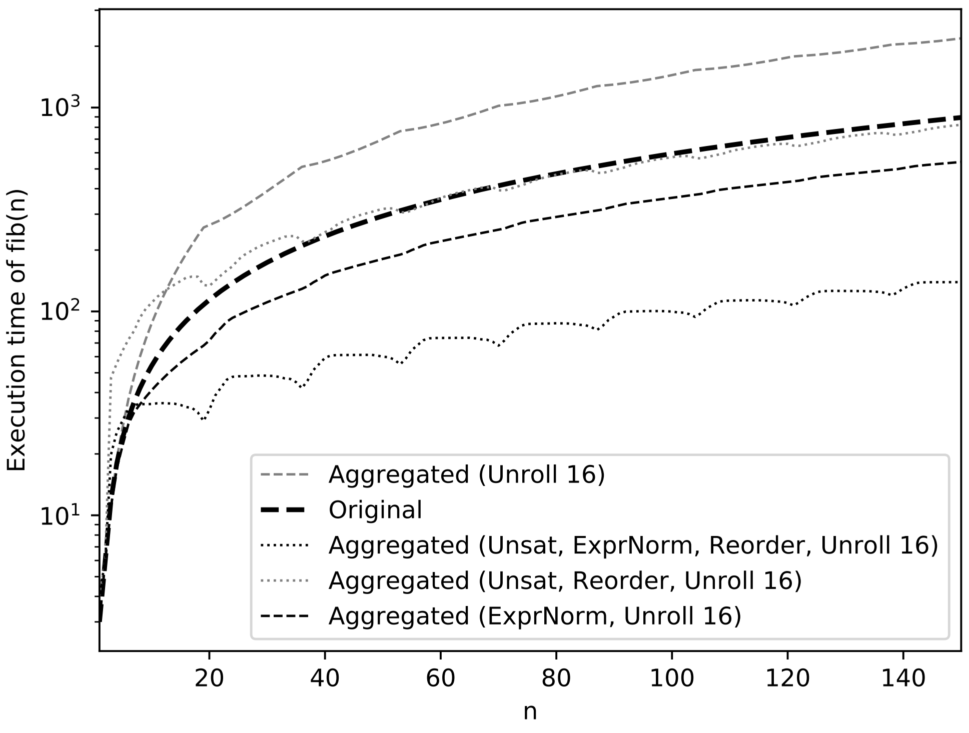


Figure 3.21: Execution times of different combinations of optimisation techniques, all based on 16 loop unrollings.

In contrast to the comparison in Figure 3.20 where all of our optimisations are enabled, Figure 3.21 shows the effect of different optimisations on the resulting execution time. The fastest execution time is achieved when all optimisations are enabled.

Predicate reordering alone has a slightly negative effect as it introduces infeasible paths and might lead to a redundant evaluation of predicates. Moreover, the infeasible path elimination alone has no effect as the original input program does not contain any infeasible paths. The combination of these two optimisations is beneficial as the infeasible path elimination removes paths in the ADDs which are unnecessarily introduced by our predicate reordering. Expression normalisation turns out to be essential for the Fibonacci program as it simplifies the expanded expressions resulting from loop unrolling. This is reflected in Figure 3.21 as the execution times of variants without expression normalisation are higher compared to those where our expression normalisation is enabled.

In summary, we have observed that

- loop urolling is crucial to enable the full potential of the considered optimisations with a large number of unrollings being preferable,
- expression normalisation accelerates the execution because it simplifies expanded expressions that result from loop unrolling, and
- the combination of predicate reordering and infeasible path elimination is beneficial, whereas individually, these techniques are only helpful in specific cases, e.g. when the original input program already contains infeasible paths.

Where standard compilers struggle to optimise the example Fibonacci program (Fig. 3.12), our optimisation approach allows for a drastic speedup of more than an order of magnitude compared to the original program.

The decomposition of the original program into ADDs with an ED as its auxiliary data structure constitutes a true generalisation of the previously discussed Random Forest transformation [3]$_{\text{AP}}$. This generalisation optimises while programs, a representative for general-purpose programming languages, and applies a very similar transformation. Any Random Forest can naturally be expressed in the form of an equivalent acyclic while program such that the resulting ADD is the exact same with both transformations — the Random Forest-specific transformation and the general while program transformation. In this sense, the here discussed compilation approach does not only build upon previous ideas but is, in fact, a true generalisation.

The discussed Fibonacci program was chosen as an example that appears hard to optimise. As such it gives a good impression of the potential for this new compilation paradigm but is by no means an extensive evaluation. The work on this and other ADD-based program optimisations continues and some particularly interesting aspects will be discussed in the corresponding Chapter 5.

4 ADD-Lib: A Highly Flexible Framework for Decision Diagrams and Code Generation

The ADD-Lib [93] is a framework for decision diagrams and constitutes an integral part of this dissertation. It is the key enabler for all implementations and evaluations that complement the scientific contributions. The framework is designed with flexibility and ease of use in mind and, as such, seamlessly adapts to the various algebraic structures.

There exist many implementations of the standard algorithms for Binary Decision Diagrams (BDDs), Algebraic Decision Diagrams (ADDs), and even for Zero-Suppressed Decision Diagrams (ZDDs) [94, 105, 106][107, 108, 109]. However, these realisations lack flexibility at their core: The underlying algebraic structures, e.g. the standard Boolean logic, are hard coded — even for ADDs. In CUDD [94], e.g., ADDs are limited to real numbers[1] and standard arithmetic operations. The framework is, in fact, so rigid that a change of the algebraic structure imposes a non-trivial adaptation of the library's core at compile time — a change that would affect major parts of the code base.

With the ADD-Lib, we provide a framework that overcomes these limitations: Our decision diagram framework is highly flexible and puts emphasis on the interchangeability of the underlying algebraic structure. At the same time, the implementation delegates computationally expensive operations to the well-established and robust C implementation of CUDD in a service-oriented fashion. In this way, the ADD-Lib inherits its broad range of functions and provides an easy-to-use yet flexible interface for the most common use cases.

The advantages of the ADD-Lib over other existing decision diagram libraries are threefold:

- The algebraic structure underlying the various ADDs is interchangeable in a service-oriented fashion. With only a few lines of code, it is possible to fully define a new ADD variant that takes advantage of the extensive collection of ADD algorithms in the ADD-Lib.
- For analysis, debugging, and presentation purposes, the ADD-Lib comes with visualisation capabilities that allow its users to get an immediate impression of their in-memory diagrams.

[1]Real numbers are represented as `double` values.

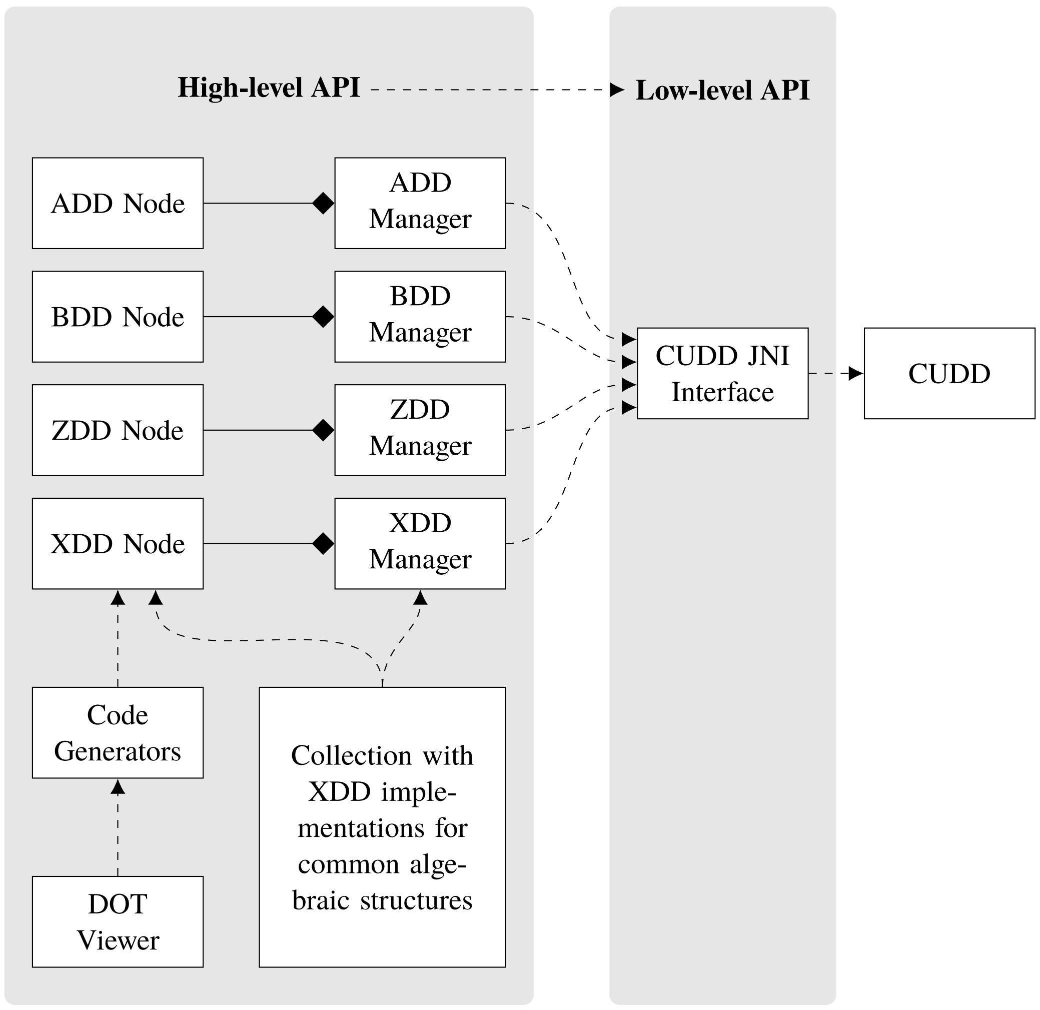


Figure 4.1: Architecture overview of the ADD-Lib.

- Particularly important in the context of program optimisation, the ADD-Lib comes with code generation capabilities that allow embedding optimal decision diagrams in various general-purpose programming languages.

Figure 4.1 shows an overview of the ADD-Lib's architecture. The most important parts of the API are the `XDD` nodes and the respective `XDDManagers`. They represent ADDs in the most general sense and allow for derived ADD implementations with custom algebraic structures. Individual nodes, or functions, are exposed through the `XDD` node interface and provide methods to invoke all of the defined operations directly. The `XDDManager` conceptually contains all of the `XDD` nodes and ensures their uniqueness and predicate order. It also provides means to create constants and other primitives.

Many of the common algebraic structures are already implemented in the ADD-Lib. This extensive example set of ADD implementations is based on the `XDD` node and `XDDManager` interfaces through inheritance.

The library comes with code generators for various languages. These are completely independent of the `XDD` implementations and treat them as data. The visualisation tools are similar in this sense and utilise the DOT code generator for their renderings.

The ADD-Lib also exposes three other kinds of decision diagrams: `ADDs`, `BDDs`, and `ZDDs`, all of which are tied to a fixed algebraic structure. The `ADD` interface represents ADDs as they are implemented in CUDD with arithmetic operations on floating point values, while `BDDs` and `ZDDs` both operate on the Boolean algebra. These diagram types are exposed for performance reasons only as they are directly implemented in CUDD and do not require any bookkeeping overhead. Each of these diagrams can be translated to an `XDD` with the respective algebraic structure.

At its core, the ADD-Lib is based on CUDD [94]. The low-level API exposes a Java interface that closely resembles that of the CUDD library. Using this interface gives access to more functions of the underlying implementation at the cost of complexity. In contrast, the high-level API is more convenient to use, comes with code generators and visualisation, and delegates all computationally expensive tasks to CUDD through the low-level API.

4.1 Algebra as a Service: Algebraic Decision Diagrams in the ADD-Lib

The definition of a new algebraic structure requires no more than a specification of its carrier set and the associated operations. The algebraic structure can be defined directly on the later co-domain of the ADDs (Chap. 2). That very definition is then plugged into the algorithms for the corresponding operations on the decision diagram data structure.

In exactly the same way, the ADD-Lib realises its ADDs. Based on a concise description of the algebraic structure, it provides the corresponding ADDs with the full range of functions on them. No additional effort is required by the user and ADD operations, visualisation, and code generation work seamlessly with any custom algebraic structure.

Consider, for example, the list monoid that was used in $[3]_{\mathrm{AP}}$. The definition of this algebraic structure is simple and defines the monoid with its join operation $\circ$ and a neutral element ϵ.

```java
public class ListMonoidDDManager extends MonoidDDManager<List<String>> {

  @Override
  protected List<String> neutralElement() {
    return new ArrayList<>();
  }

  @Override
  protected List<String> join(List<String> lhs, List<String> rhs) {
    ArrayList<String> joined = new ArrayList<>();
    joined.addAll(lhs);
    joined.addAll(rhs);
    return joined;
  }
}
```

Figure 4.2: Java code to define ADDs over a List Monoid (Def. 19) in the ADD-Lib.

Definition 19 (List Monoid) *The List Monoid $\mathcal{A}_l = (\Sigma^*, \circ, \epsilon)$ over an alphabet Σ is defined with*

$$\begin{aligned} \textit{concatenation} \quad & [\, a_0, \ldots, a_{n-1} \,] \circ [\, b_0, \ldots, b_{m-1} \,] := [\, a_0, \ldots, a_{n-1}, b_0, \ldots, b_{m-1} \,] \\ \textit{and the neutral element} \quad & \epsilon := [\ ]. \end{aligned}$$

The definition in the ADD-Lib is exactly as simple and can be realised with only a few lines of Java code. Figure 4.2 shows the complete code needed to implement ADDs over the List Monoid (Def. 19).

To facilitate the implementation of a wide range of custom algebraic structures, the ADD-Lib comes with templates for the most common use cases. The categorisation into

- groups and group-like structures,
- rings and ring-like structures, and
- various lattices and logics

allows us to determine the required operations already at compile time. For example, a Boolean lattice will require conjunction, disjunction, and negation as well as the distinguished 0 and 1 elements.

Technically, these templates are realised as abstract classes that enforce a concrete implementation for the required elements per algebraic structure. The `ListMonoidDDManager` in Figure 4.2, e.g., extends the super class for monoids, `MonoidDDManager`. Already at compile time, this allows us to ensure that the List Monoid indeed implements a join operation as well as a neutral element.

On a more technical note, the `ListMonoidDDManager` acts as a container for the unique nodes of the ADDs over the List Monoid. It provides interfaces to construct, manipulate and compose these ADDs and also manages the required memory. Using this factory-like construction ensures the immutability and finally the key properties of decision diagrams: canonicity and size-optimality.

The List Monoid is just one example and many others require no more code to realise in the ADD-Lib. All of the logics discussed in Chapter 2 can be realised in the exact same way. In fact, the ADD-Lib comes with a wide range of predefined algebraic structures ranging from various logics, to vector spaces and even to power set lattices.

4.2 Decision Diagram Visualisation

Besides being an extremely flexible framework for ADDs, the ADD-Lib comes with powerful tools that facilitate experimentation with various forms of decision diagrams. A crucial aspect of this is *visualisation*. Exploiting the graph nature of decision diagrams, the ADD-Lib is able to give visual insights into the decision diagram structure. A visual presentation helps understanding the often complex structure of these diagrams. This helps, in particular, to present ideas, not least for the figures in publications and in this dissertation.

The instant visualisation features of the library are also helpful for debugging. They allow to inspect intermediate ADDs as well as the final result and, in this way, help verifying that transformations are implemented as intended.

There are generally two ways, to render decision diagrams with the ADD-Lib:

- The `DotViewer` is a tool to quickly render the in-memory decision diagram. From within the user code, the viewer is invoked with a single line of code and displays the diagram in a dedicated window. In this way, the tool allows for quick and easy debugging. The viewer comprises functions for navigating in the diagram and also to save the diagram as a file.

- Alternatively, the ADD-Lib can export decision diagrams in various formats, including image formats and also DOT code that is suitable for subsequent processing. This feature is what we use for most visualisations needed in publications. In this way, chosen aspects of the diagrams could be highlighted and the formatting could be further customised beyond what is possible automatically. All of these export functions are in fact also accessible through the `DotViewer`'s interface.

Figure 4.3 shows a screenshot of the `DotViewer` rendering an example ADD from [3]$_{\text{AP}}$. The ADD operates on the List Monoid (Def. 19) and predicts a flower's species based on four measurements. This example diagram is relatively small and can be inspected with the viewer based on the in-memory data structure. The diagram's terminal nodes hold the lists of strings in this case as can be seen, e.g., in the highlighted terminal node.

4.3 Code Generation: From Decision Diagrams to Executable Code

Another extremely useful feature in the context of program optimisation that sets the ADD-Lib apart from other decision diagram frameworks is its code generation capability. With this, we close the circle back to the original idea of Binary Decision Programs (BDPs) [48], however,

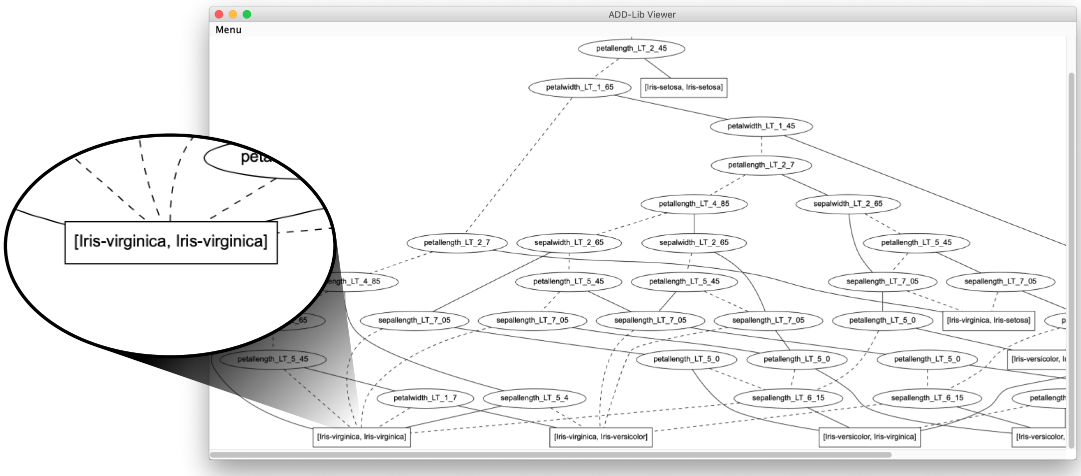


Figure 4.3: Screenshot of the ADD-Lib viewer showing an Iris species predicting ADD.

with some notable improvements through the evolution of decision diagrams: Most notably, the generated programs inherit the key properties of ADDs: They are

- optimal in the same sense as ADDs are, i.e. they are canonical and size-optimal, and
- represent functions with an arbitrary co-domain of the form $\mathbb{B}^n \to \mathcal{A}$.

The various code generators take an ADD and generate its function implementation in one of many supported general-purpose programming languages, e.g. C/C++, C#, Java, Python, etc. With this, the generated code can be (i) interpreted by the language's corresponding interpreter, or (ii) compiled and executed at native speed. In this way, we can take advantage not only of the inherent running time optimisation of ADDs but additionally of native compilation speedups.

All of the ADD-Lib's code generators follow a common pattern: They generate the graph structure of decision diagrams as *goto* programs where every node is rendered as one label in the resulting implementation. Although generally perceived as an anti-pattern in programming [49], *goto* programs have one important advantage: they allow for general graph-like program structures other than, e.g., while programs. For this reason, they are ideally suited to embed the structure of decision diagrams in program code without any loss. While *goto* programs are generally regarded as hard to ready, this property is irrelevant in the generated code. When reasoning about one of these program, we would focus on the input specification, i.e. the format that a user actually manipulates, not the automatically generated and highly optimised and compiled code.

Some programming languages like Java and Python have abandoned goto statements altogether. In this way, they prevent programmers from writing unreadable *goto* programs. While this is generally not a bad idea, the downside of the decision is that some performance optimisations are no longer possible without additional variables, and thus some degree of indirection. It may be for this reason that programming languages avoiding goto statements are, unlike C and C++, not primarily aimed at performance.

```cpp
std::vector<std::string> predictIrisSpecies() {
  goto eval140349844102272;

  eval140349844102272:
  if (petallength_LT_2_45()) goto eval140349844099616;
  else goto eval140349844102208;

  ...

  eval140349844100416:
  return {"Iris-virginica", "Iris-virginica"};

  ...
}
```

Figure 4.4: Excerpt of generated C++ code for the previously seen Iris species predicting ADD (Fig. 4.3).

An alternative pattern for code generation is one additional state variable that keeps track of the current state. This way, the missing goto statement is essentially simulated and importantly the computational advantage of the original decision diagram is fully preserved.

Figure 4.4 shows an excerpt of the generated C++ code for the running example of an aggregated ADD. There are, of course, many more internal and terminal nodes rendered than seen in this excerpt. The code is hardly readable but highly optimised for running time, both results of the ADD structure. Interestingly, the generated *goto* programs very closely resemble the idea of BDPs [48] but, of course, they fully embrace the evolution of decision diagrams since then [50, 34, 28].

The ADD-Lib comes with a collection of code generators for a multitude of target languages. Most notably, general-purpose languages like Java, C#, Python, JavaScript, C, and C++ are supported but code generators also exist for some export formats like images and DOT code. As an open-source project, the ADD-Lib encourages its users to add their own target languages and even comprises a code generator generator.

5 Future Work

The exploration of Algebraic Decision Diagrams (ADDs) as an Intermediate Representation (IR) for program optimisation in the three program domains yielded some impressive results and motivates future research: The ADD-based Domain-Specific Languages (DSLs) will, e.g., benefit from semantic predicate treatment, other machine-learned programs are likely to benefit from ADD-based optimisations, and the exploration of the IR's potential in the context of general-purpose programs must be analysed on an extensive benchmark suite. In this chapter, we will discuss open questions and opportunities for future research per the three program domains.

5.1 Domain-Specific Languages

The starting point for ADD-based program optimisation were DSLs tailored to the targeted IR, the ADDs. We have explored their potential in the context of an email classification system [6, 7]$_{\text{AP}}$ with a fully functional implementation. The basis for decision making was a Boolean description of the incoming emails. Each of these predicates could be used in the modelled ADDs and the resulting data structure is optimal under the assumption that these predicates are independent of one another. However, in reality, this might not be the case which can lead to infeasible paths in the data structure. The effect of this is similar to that observed in the Random Forest transformation: unnecessarily large ADDs and potentially redundant predicate occurrences. The solution is, again, the elimination of infeasible paths [110][3, 4]$_{\text{AP}}$.

In this case, the interplay of predicates may not be as obvious. The implications among them are not given and cannot be derived from a known theory. As a consequence, a compiler will not be able to resolve resulting infeasibilities automatically without additional information. The necessary information can easily be given through the predicate language where a user could incorporate his or her expert knowledge directly in the model. These can be simple implications between predicates or possibly more complex relations. On the basis of such a feasibility model, the compiler can then resolve infeasible paths in the ADDs in a similar fashion as was successfully done in the Random Forest transformation [3]$_{\text{AP}}$ and in the while transformation [4]$_{\text{AP}}$, respectively.

The second rule-based DSL, MiAamics, can equally benefit from this improvement with the anticipated results being the exact same. Again, the implications among predicates must be given as part of the input format. In the case of MiAamics, some standard theories may also be incorporated in the language to simplify its use.

Both DSLs were evaluated with examples and automatically generated input, however, their application in industrial-size decision problems would help to further evaluate strengths and weaknesses of the approach. Recommender systems are a prime area for application here, but really any performance-critical decision procedure is a candidate for another case study. Enabling domain experts to capture their ideas in a mindset-oriented language and apply fully automatic and holistic optimisations during code generation can prove to be a powerful tool.

Another aspect of interest in the context of model-driven development [68, 59] is the DSLs' embedding into the ecosystem of DSLs. At this point, some manual work in the form of adapters is needed to bridge the gap between these DSLs. The seamless integration of our ADD-based DSLs is a topic for future research. It is, however, not specific to these DSLs but rather a general problem in the Cinco context [59][63].

5.2 Machine-Learned Models

As a popular classifier, Random Forests have served as a representative for machine-learned programs. There are, however, many more classifiers, some of which may benefit from very similar optimising transformations. Interesting classifiers are, in particular, those that allow for a semantics-preserving transformation to ADDs, i.e. those that allow for discretisation by means of predicates.

The Random Forest transformation can be naturally generalised to Decision Jungles [111], a classifier that learns a single Directed Acyclic Graph (DAG) instead of many trees. In contrast to ADDs, this DAG is not constrained in its predicate order and it is solely motivated from a machine-learning perspective. Thus, the generalised transformation will still restructure the input program entirely. While the transformation is straightforward, the resulting effect in terms of running time and size might differ drastically.

The dominating trends in machine learning today are, of course, Deep Neural Networks (DNNs) [112, 113]. These neural networks represent highly non-linear functions and are learned with the help of large datasets. Their success in many important domains, in computer vision and beyond, has made them a vivid area of research [114, 115, 116, 117]. DNNs are computationally intensive so that it is well established to use Graphics Processing Units (GPUs) for their learning and inference at runtime. Setting the computational effort of learning aside, the evaluation of these functions at runtime is a problem that could potentially be targeted with a suitable model transformation along the lines of our Random Forest transformation. DNNs are not only of interest to many researchers and practitioners equally, but they involve highly performance critical tasks. A challenge will be the continuous nature of these models which makes it difficult to transform them to a discrete structure like ADDs.

A promising variant of DNNs are Binary Neural Networks (BNNs) [118] where each perceptron can yield only one of two values. The discrete values of their perceptrons translate well to the binary nature of predicates in ADDs allowing for a semantics-preserving transformation. The main challenge here is to tame the size of the resulting ADDs, may it be through the elimination of infeasible paths or a simplification of the model. Ideally, the semantics would be fully preserved but in cases where this is not possible, the simplification must be justified with experiments on actual data in a fashion machine learning researchers are well familiar with.

This line of research on semantics-preserving transformations of BNNs has already started with the help of Jan Linden and Jan Feider. Building on our experience and also on previous work by [119, 120, 121, 122], we were able to transform small networks trained on MNIST dataset for handwritten digits [123] into semantically equivalent ADDs. We were able to evaluate the resulting ADD about 20 times faster than the original neural network. These initial impressions motivate further investigation of this application area.

5.3 General-Purpose Programming Languages

The work on general-purpose programming languages is a radically different compilation paradigm. The initial experiments with the iterative Fibonacci implementation are very promising but an extensive evaluation with more representative benchmark programs is needed. Ideally, this benchmark suite comprises programs of industrial relevance.

A particularly promising class of programs are Programmable Logic Controller (PLC) programs [124]. These programs are acyclic in nature and would therefore not rely on the choice of good cut points. The work on this has already started with the help of Alnis Murtovi, Marc Jasper, and David Schmidt.

For a broad coverage of benchmarks, the new compilation paradigm must be implemented in a compiler for some widely used general-purpose programming language. Ideally, this would be based on LLVM, covering a range of different programming languages. This would allow to target, e.g., C for which benchmark programs are widely available.

The current evaluation is based on a machine-independent measure which is a good way to evaluate the potential on a conceptual level without relying on details of implementation or even the executing machine. In addition to this, an evaluation based on wall clock time would give an insight into the interplay of this program optimisation approach and machine-specific details. These measurements would take into account the varying costs among arithmetic operations as well as machine-specific caching behaviour.

The current optimisation technique optimises on the bases of acyclic fragments. A more global extension following the paradigm would also allow for inter-program fragment optimisations. Cut points would still remain the anchor points of the program but rather than *jumping* only between cut points and their successor, arbitrary paths could be evaluated at compile time. This allows, for example, to speed up loops even more without considering every possible number of iterations. For example, 0, 1, 2, 4, 8, etc. iterations could be selected and only the corresponding paths aggregated. The anticipated result are aggregated path ADDs of smaller size that, at the same time, can still jump over multiple iterations of the loop. In fact, this generalisation could allow for arbitrary shortcuts in the original program.

With the aggregation of entire program fragments into the three complementing IRs, the approach allows for extensive parallelisation. In order to evaluate the ADD, all predicates along the taken path must be evaluated, meaning that the corresponding subset of the Expression DAG

(ED) must be available. While the specific subset is unclear, the evaluation can also be performed speculatively for all the predicates. This can be done in parallel per layer of the ED, raising the question if and how this speculative evaluation can effectively take advantage of accelerators like GPUs. If successful, this transformation would restructure the input program entirely and it would effectively implement an automatic parallelisation of iteratively specified programs.

6 Conclusion

In the modern spirit of Intermediate Representation (IR) families in compilers, as encouraged by frameworks like MLIR, this dissertation explored the potential of Algebraic Decision Diagrams (ADDs) as an IR for (domain-specific) program optimisation. We presented successful applications in a variety of quite different program domains. Exploiting the data structure's inherent optimality in size and depth, i.e. running time, we were able to transfer these traits to quite different program structures and use cases. In this way, the corresponding optimising compilers could, quite effectively, optimise the programs at hand and on the bases of the *right* representation, in this case on the bases of ADDs.

Initially, we targeted graphical Domain-Specific Languages (DSLs) that are structurally quite similar to ADDs which allows for a straightforward transformation of the program to its IR and, as a consequence, effective running time optimisation of the decision procedure [6, 7]$_{\text{AP}}$. There is, however, great potential beyond this simple case: We also targeted and extended a textual DSL, MiAamics, that is in structure quite different from ADDs [8]$_{\text{AP}}$. This rule-based DSL was initially designed for recommender systems. With MiAamics, we were able to rapidly evaluate extremely large systems of rules. Although structurally quite different from ADDs, ADD-based aggregation and optimisation was quite successful — a success that is likely transferrable to many other discrete decision DSLs. In both cases, graphical and textual DSLs, we were able to transform the input program to an equivalent ADD and subsequently to an implementation through code generation.

A program domain that benefits to the extreme from ADD-based optimisation are Random Forests [3, 5]$_{\text{AP}}$. Here, we targeted a well-established domain of programs. The ADD-based aggregation also transforms this input program to a semantically equivalent ADD and allows for an efficient implementation through code generation. The Random Forest's running time and size grow linearly with the number of trees. We were able to achieve impressive speedups here: Aggressive aggregation allows us to reduce both, running time and size, by multiple orders of magnitude.

Although the initial motivation was to reduce running time and size, the developed aggregation technique is also an extremely useful tool for a different discipline: explainability. With the aggregated ADD, we were able to solve three explainability problems for Random Forests:

model explanation, class characterisation, and outcome explanation.

The success of the Random Forest transformation also inspired its generalisation to general-purpose programming languages $[4]_{\mathrm{AP}}$. The *while* language serves as a representative and we were able to fully generalise the concepts and compile these programs in the very same fashion. Also here, we were able to achieve surprising reductions in running time, illustrated for a seemingly optimal implementation of the Fibonacci numbers.

The aggregation of entire program fragments and their representation by means of three complementing IRs holds great potential for parallelisation. A highly speculative and parallel execution built on this idea is subject of an already started follow-up research project at the TU Dortmund. The project will explore the realisation of such a radically different compilation to target not only Central Processing Units (CPUs) but also accelerators like Graphics Processing Units (GPUs).

Of course, ADDs are not the solution to every problem. The most pressing issue is their size that can grow exponentially with the number of predicates. In some cases, ADDs can outright explode. However, we have seen in multiple cases that this issue can be tamed with the right predicate order and the right treatment of their predicates' semantics.

This dissertation has only started to explore the ADDs potential as an IR and has already inspired fellow researchers to explore their potential further. With the ADD-Lib, they are given an extensive toolkit and can take full advantage of our work to date. ADDs have already proven their potential as an IR for (domain-specific) program optimisation in various program domains — a success that can likely be extended to further program domains in the future.

References

[1] F. Gossen and B. Steffen, “Algebraic aggregation of random forests: Towards explainability and rapid evaluation,” *International Journal on Software Tools for Technology Transfer*, 2021. accepted.

[2] F. Gossen, T. Margaria, and B. Steffen, “Towards explainability in machine learning: The formal methods way,” *IT Professional*, vol. 22, no. 04, pp. 8–12, 2020.

[3] F. Gossen and B. Steffen, “Large random forests: Optimisation for rapid evaluation,” *CoRR*, vol. abs/1912.10934, 2019.

[4] F. Gossen, M. Jasper, A. Murtovi, and B. Steffen, “Aggressive aggregation: a new paradigm for program optimization,” *CoRR*, vol. abs/1912.11281, 2019.

[5] F. Gossen, A. Murtovi, P. Zweihoff, and B. Steffen, “Add-lib: Decision diagrams in practice,” *CoRR*, vol. abs/1912.11308, 2019.

[6] B. Steffen, F. Gossen, S. Naujokat, and T. Margaria, *Language-Driven Engineering: From General-Purpose to Purpose-Specific Languages*, pp. 311–344. Springer International Publishing, 2019.

[7] F. Gossen, T. Margaria, A. Murtovi, S. Naujokat, and B. Steffen, “Dsls for decision services: A tutorial introduction to language-driven engineering,” in *Leveraging Applications of Formal Methods, Verification and Validation. Modeling*, pp. 546–564, Springer International Publishing, 2018.

[8] F. Gossen and T. Margaria, “Generating optimal decision functions from rule specifications,” *Electronic Communications of the EASST*, vol. 74, 2017.

[9] H. Hungar, B. Steffen, and T. Margaria-Steffen, “Device for generating selection structures, for making selections according to selection structures and for creating selection.” United States Patent Application Publication, 2013. Appl. No.: 13/650,680.

[10] C. Lattner and V. Adve, “Llvm: a compilation framework for lifelong program analysis transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, 2004.

[11] “The LLVM compiler infrastructure.” `https://llvm.org/`. Accessed: 2020-02-16.

[12] M. Paleczny, C. Vick, and C. Click, “The java hotspot tm server compiler,” in *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium*, vol. 1, 2001.

[13] “GCC, the GNU compiler collection.” `https://gcc.gnu.org/`. Accessed: 2020-02-15.

[14] “libFIRM: Optimization and machine code generation.” `https://pp.ipd.kit.edu/firm/`. Accessed: 2020-02-15.

[15] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Global value numbers and redundant computations,” in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 12–27, 1988.

[16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

[17] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau, “Simple and efficient construction of static single assignment form,” in *Compiler Construction* (R. Jhala and K. De Bosschere, eds.), pp. 102–122, Springer Berlin Heidelberg, 2013.

[18] E. Morel and C. Renvoise, “Global optimization by suppression of partial redundancies,” *Commun. ACM*, vol. 22, no. 2, pp. 96–103, 1979.

[19] B. Steffen, “Property-oriented expansion,” in *Static Analysis, Third International Symposium, SAS’96, Aachen, Germany, September 24-26, 1996, Proceedings*, pp. 22–41, 1996.

[20] J. Knoop, O. Rüthing, and B. Steffen, “Partial dead code elimination,” in *Proceedings of the ACM SIGPLAN’94 Conference on Programming Language Design and Implementation (PLDI)*, pp. 147–158, 1994.

[21] J. Knoop, O. Rüthing, and B. Steffen, “Optimal code motion: Theory and practice,” *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 4, pp. 1117–1155, 1994.

[22] B. Steffen and J. Knoop, “Finite constants: Characterizations of a new decidable set of constants,” *Theor. Comput. Sci.*, vol. 80, no. 2, pp. 303–318, 1991.

[23] A. V. Aho, R. Sethi, and J. D. Ullman, “Compilers, principles, techniques,” *Addison wesley*, vol. 7, no. 8, p. 9, 1986.

[24] “MLIR unifies the infrastructure for high-performance ML models in TensorFlow..” `https://www.tensorflow.org/mlir`. Accessed: 2020-02-15.

[25] C. Lattner, J. A. Pienaar, M. Amini, U. Bondhugula, R. Riddle, A. Cohen, T. Shpeisman, A. Davis, N. Vasilache, and O. Zinenko, "MLIR: A compiler infrastructure for the end of moore's law," *CoRR*, vol. abs/2002.11054, 2020.

[26] "Tensorflow: An end-to-end open source machine learning platform." `https://www.tensorflow.org/`. Accessed: 2020-02-16.

[27] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic decision diagrams and their applications," in *Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design*, IEEE Computer Society Press, 1993.

[28] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebric decision diagrams and their applications," *Formal Methods in System Design*, vol. 10, 1997.

[29] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi, "A survey of methods for explaining black box models," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 93:1–93:42, 2019.

[30] F. Gossen, "Bayesian recognition of human identities from continuous visual features for safe and secure access in healthcare environments," 2015.

[31] F. Gossen and T. Margaria, "Comprehensible people recognition using the kinect's face and skeleton model," in *2016 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, pp. 1–6, IEEE, 2016.

[32] B. Steffen, T. Margaria, and V. Braun, "Personalized electronic commerce services," in *IFIP WG 7.3 8th Int. Conference on Telecommunication Systems Modeling and Analysis*, 2000.

[33] B. Steffen, T. Margaria, and V. Braun, "Coarse-granular model checking in practice," in *Model Checking Software* (M. Dwyer, ed.), pp. 304–311, Springer Berlin Heidelberg, 2001.

[34] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, 1986.

[35] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 2018.

[36] K. L. McMillan, "Symbolic model checking," in *Symbolic Model Checking*, pp. 25–60, Springer, 1993.

[37] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang, "Symbolic model checking: 1020 states and beyond," *Information and computation*, vol. 98, no. 2, pp. 142–170, 1992.

[38] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli, “Logic verification using binary decision diagrams in a logic synthesis environment,” in *[1988] IEEE International Conference on Computer-Aided Design (ICCAD-89) Digest of Technical Papers*, pp. 6–9, IEEE, 1988.

[39] R. M. Sinnamon and J. D. Andrews, “Fault tree analysis and binary decision diagrams,” in *Proceedings of 1996 Annual Reliability and Maintainability Symposium*, pp. 215–222, IEEE, 1996.

[40] A. Shih, A. Choi, and A. Darwiche, “Compiling bayesian network classifiers into decision graphs,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 7966–7974, 2019.

[41] B. Steffen and A. Murtovi, “M3c: Modal meta model checking,” in *Formal Methods for Industrial Critical Systems* (F. Howar and J. Barnat, eds.), pp. 223–241, Springer International Publishing, 2018.

[42] T. Tegeler, A. Murtovi, M. Frohme, and B. Steffen, *Product Line Verification via Modal Meta Model Checking*, pp. 313–337. Springer International Publishing, 2019.

[43] S.-i. Minato, “Zero-suppressed bdds for set manipulation in combinatorial problems,” in *Proceedings of the 30th International Design Automation Conference*, DAC ’93, pp. 272–277, Association for Computing Machinery, 1993.

[44] R. E. Bryant, “Chain reduction for binary and zero-suppressed decision diagrams,” in *Tools and Algorithms for the Construction and Analysis of Systems* (D. Beyer and M. Huisman, eds.), pp. 81–98, Springer International Publishing, 2018.

[45] T. Kam, T. Villa, R. K. Brayton, and A. L. Sangiovanni-vincentelli, “Multi-valued decision diagrams: theory and applications.,” *Multiple-Valued Logic*, vol. 4, pp. 9–62, 1998.

[46] S. Blom and J. van de Pol, “Symbolic reachability for process algebras with recursive data types,” in *Theoretical Aspects of Computing - ICTAC 2008* (J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigun, eds.), pp. 81–95, Springer Berlin Heidelberg, 2008.

[47] T. Sasao, “Ternary decision diagrams. survey,” in *Proceedings 1997 27th International Symposium on Multiple- Valued Logic*, pp. 241–250, 1997.

[48] C. Y. Lee, “Representation of switching circuits by binary-decision programs,” *Bell System Technical Journal*, vol. 38, no. 4, pp. 985–999, 1959.

[49] E. W. Dijkstra, “Letters to the editor: Go to statement considered harmful,” *Commun. ACM*, vol. 11, no. 3, pp. 147–148, 1968.

[50] S. B. Akers, “Binary decision diagrams,” *IEEE Trans. Comput.*, vol. 27, no. 6, pp. 509–516, 1978.

[51] C. E. Shannon, “A symbolic analysis of relay and switching circuits,” 1937.

[52] G. Boole, *The mathematical analysis of logic: being an essay towards a calculus of deductive reasoning*. Macmillan, Barclay, & Macmillan, 1847.

[53] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transforms for large boolean functions with applications to technology mapping," in *Proceedings of the 30th International Design Automation Conference*, DAC '93, ACM, 1993.

[54] E. M. Clarke, K. L. Mcmillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transforms for large boolean functions withapplications to technology mapping," *Form. Methods Syst. Des.*, vol. 10, 1997.

[55] M. Bergmann, *An Introduction to Many-Valued and Fuzzy Logic: Semantics, Algebras, and Derivation Systems*. Cambridge University Press, 2008.

[56] V. Novák, I. Perfilieva, and J. Mockor, *Mathematical principles of fuzzy logic*, vol. 517. Springer Science & Business Media, 2012.

[57] M. Voelter, "Fusing modeling and programming into language-oriented programming," in *International Symposium on Leveraging Applications of Formal Methods*, pp. 309–339, Springer, 2018.

[58] L. C. Kats and E. Visser, "The spoofax language workbench: rules for declarative specification of languages and ides," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pp. 444–463, 2010.

[59] S. Naujokat, M. Lybecait, D. Kopetzki, and B. Steffen, "Cinco: a simplicity-driven approach to full generation of domain-specific graphical modeling tools," *International Journal on Software Tools for Technology Transfer*, vol. 20, no. 3, pp. 327–354, 2018.

[60] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt, "A programmable programming language," *Commun. ACM*, vol. 61, no. 3, pp. 62–71, 2018.

[61] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[62] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 307–309, 2010.

[63] "CINCO SCCE meta tooling framework." `https://cinco.scce.info/about/`. Accessed: 2020-02-18.

[64] "Another tool for language recognition." `https://www.antlr.org/`. Accessed: 2020-02-18.

[65] "Language engineering for everyone." `https://www.eclipse.org/Xtext/`. Accessed: 2020-02-18.

[66] “MPS: Meta programming system.” `https://www.jetbrains.com/mps/`. Accessed: 2020-02-18.

[67] C. Kubczak, T. Margaria, B. Steffen, C. Winkler, and H. Hungar, *An Approach to Discovery with miAamics and jABC*, pp. 217–234. Springer US, 2009.

[68] S. Boßelmann, M. Frohme, D. Kopetzki, M. Lybecait, S. Naujokat, J. Neubauer, D. Wirkner, P. Zweihoff, and B. Steffen, “Dime: A programming-less modeling environment for web applications,” in *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications* (T. Margaria and B. Steffen, eds.), pp. 809–832, Springer International Publishing, 2016.

[69] F. Abel, I. I. Bittencourt, N. Henze, D. Krause, and J. Vassileva, “A rule-based recommender system for online discussion forums,” in *Adaptive Hypermedia and Adaptive Web-Based Systems* (W. Nejdl, J. Kay, P. Pu, and E. Herder, eds.), pp. 12–21, Springer Berlin Heidelberg, 2008.

[70] B. Bouihi and M. Bahaj, “Ontology and rule-based recommender system for e-learning applications,” *International Journal of Emerging Technologies in Learning (iJET)*, vol. 14, no. 15, pp. 4–13, 2019.

[71] T. Margaria and B. Steffen, “Simplicity as a driver for agile innovation,” *Computer*, vol. 43, no. 6, pp. 90–92, 2010.

[72] G. Linden, B. Smith, and J. York, “Amazon.com recommendations: item-to-item collaborative filtering,” *IEEE Internet Computing*, vol. 7, no. 1, pp. 76–80, 2003.

[73] B. Smith and G. Linden, “Two decades of recommender systems at amazon.com,” *IEEE Internet Computing*, vol. 21, no. 3, pp. 12–18, 2017.

[74] C. A. Gomez-Uribe and N. Hunt, “The netflix recommender system: Algorithms, business value, and innovation,” *ACM Trans. Manage. Inf. Syst.*, vol. 6, no. 4, pp. 13:1–13:19, 2015.

[75] J. Davidson, B. Liebald, J. Liu, P. Nandy, T. Van Vleet, U. Gargi, S. Gupta, Y. He, M. Lambert, B. Livingston, and D. Sampath, “The youtube video recommendation system,” in *Proceedings of the Fourth ACM Conference on Recommender Systems*, RecSys ’10, pp. 293–296, ACM, 2010.

[76] H. Zhang, I. Ganchev, N. S. Nikolov, Z. Ji, and M. O’Droma, “A hybrid service recommendation prototype adapted for the UCWW: A smart-city orientation,” *Wireless Communications and Mobile Computing*, vol. 2017, 2017.

[77] C. Kubczak, “Entwicklung einer verteilten umgebung zur personalisierung von webapplikationen,” 2005.

[78] “Treelite: model compiler for decision tree ensembles.” `https://treelite.readthedocs.io`, 2017. Accessed: 2019-06-07.

[79] Facebook, "Evaluating boosted decision trees for billions of users." `https://code.fb.com/ml-applications/evaluating-boosted-decision-trees-for-billions-of-users`, 2017. Accessed: 2019-06-11.

[80] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., 4th ed., 2016.

[81] J. Browne, D. Mhembere, T. M. Tomita, J. T. Vogelstein, and R. Burns, "Forest packing: Fast parallel, decision forests," 2019.

[82] B. P. H. Kargupta, "A fourier spectrum-based approach to represent decision trees for mining data streams in mobile environments," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, 2004.

[83] P. D. Mulvaney R., "A method to merge ensembles of bagged or boosted forced-split decision trees," *IEEE Transaction on PAM*, 2003.

[84] J. G. P. Philippe J. Giabbanelli, "An algebra to merge heterogeneous classifiers," *CoRR*, 2015.

[85] A. Bonfietti, M. Lombardi, and M. Milano, "Embedding decision trees and random forests in constraint programming," in *Integration of AI and OR Techniques in Constraint Programming* (L. Michel, ed.), pp. 74–90, Springer International Publishing, 2015.

[86] A. H. Peterson and T. R. Martinez, "Reducing decision tree ensemble size using parallel decision dags," *International Journal on Artificial Intelligence Tools*, vol. 18, no. 04, pp. 613–620, 2009.

[87] A. Painsky and S. Rosset, "Lossless compression of random forests," *Journal of Computer Science and Technology*, vol. 34, no. 2, pp. 494–506, 2019-01.

[88] H. Nakahara, A. Jinguji, S. Sato, and T. Sasao, "A random forest using a multi-valued decision diagram on an fpga," in *2017 IEEE 47th International Symposium on Multiple-Valued Logic (ISMVL)*, pp. 266–271, 2017.

[89] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of Eugenics*, vol. 7, no. 7, pp. 179–188, 1936.

[90] T. K. Ho, "Random decision forests," in *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*, ICDAR '95, IEEE Computer Society, 1995.

[91] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, 2001.

[92] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, 1986.

[93] F. Gossen, A. Murtovi, J. Linden, and B. Steffen, "ADD-Lib: The java library for algebraic decision diagrams and code generation." `https://add-lib.scce.info`, 2018. Accessed: 2019-11-01.

[94] F. Somenzi, "Efficient manipulation of decision diagrams," *International Journal on Software Tools for Technology Transfer*, vol. 3, no. 2, 2001.

[95] D. Dua and C. Graff, "UCI machine learning repository." `http://archive.ics.uci.edu/ml`, 2017. Accessed: 2020-02-15.

[96] H. R. Nielson and F. Nielson, *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science, Springer, 2007.

[97] J. Knoop, O. Ruthing, and B. Steffen, "Lazy strength reduction," vol. 1, 1995.

[98] R. W. Floyd, "Assigning meanings to programs," *Proceedings of Symposium on Applied Mathematics*, vol. 19, pp. 19–32, 1967.

[99] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (C. R. Ramakrishnan and J. Rehof, eds.), pp. 337–340, Springer Berlin Heidelberg, 2008.

[100] F. Nielson and H. Riis Nielson, *Program Graphs*, pp. 1–14. Springer International Publishing, 2019.

[101] S. A. Greibach, *Theory of program structures: schemes, semantics, verification*. Springer-Verlag Berlin, 1975.

[102] M. Müller-Olm, H. Seidl, and B. Steffen, "Interprocedural herbrand equalities," in *European Symposium on Programming*, pp. 31–45, Springer, 2005.

[103] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[104] G. D. Plotkin, "A structural approach to operational semantics," 1981.

[105] T. van Dijk and J. van de Pol, "Sylvan: Multi-core decision diagrams," in *Tools and Algorithms for the Construction and Analysis of Systems* (C. Baier and C. Tinelli, eds.), pp. 677–691, Springer Berlin Heidelberg, 2015.

[106] T. van Dijk and J. van de Pol, "Sylvan: multi-core framework for decision diagrams," *International Journal on Software Tools for Technology Transfer*, vol. 19, 2016.

[107] J. Lind-Nielsen, "Buddy - a binary decision diagram package." `http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.itu.dk/research/buddy/`. Accessed: 2020-02-21.

[108] "minibdd – a minimalistic bdd library." `http://www.cprover.org/miniBDD/`. Accessed: 2020-02-21.

[109] “The bdd library.” `https://www.cs.cmu.edu/~modelcheck/bdd.html`. Accessed: 2020-02-21.

[110] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard, “Difference decision diagrams,” in *Computer Science Logic* (J. Flum and M. Rodriguez-Artalejo, eds.), pp. 111–125, Springer Berlin Heidelberg, 1999.

[111] J. Shotton, S. Nowozin, T. Sharp, J. Winn, P. Kohli, and A. Criminisi, “Decision jungles: Compact and rich models for classification,” in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS’13, pp. 234–242, Curran Associates Inc., 2013.

[112] Q. Zhang, L. T. Yang, Z. Chen, and P. Li, “A survey on deep learning for big data,” *Information Fusion*, vol. 42, pp. 146–157, 2018.

[113] L. Deng, “A tutorial survey of architectures, algorithms, and applications for deep learning,” *APSIPA Transactions on Signal and Information Processing*, vol. 3, p. e2, 2014.

[114] A. Ioannidou, E. Chatzilari, S. Nikolopoulos, and I. Kompatsiaris, “Deep learning advances in computer vision with 3d data: A survey,” *ACM Comput. Surv.*, vol. 50, no. 2, 2017.

[115] A. Brunetti, D. Buongiorno, G. F. Trotta, and V. Bevilacqua, “Computer vision and deep learning techniques for pedestrian detection and tracking: A survey,” *Neurocomputing*, vol. 300, pp. 17–33, 2018.

[116] L. Liu, W. Ouyang, X. Wang, P. W. Fieguth, J. Chen, X. Liu, and M. Pietikäinen, “Deep learning for generic object detection: A survey,” *International Journal of Computer Vision*, vol. 128, pp. 261–318, 2018.

[117] J. Han, D. Zhang, G. Cheng, N. Liu, and D. Xu, “Advanced deep-learning techniques for salient and category-specific object detection: A survey,” *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 84–100, 2018.

[118] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized neural networks,” in *Advances in neural information processing systems*, pp. 4107–4115, 2016.

[119] S. Bader, S. Hölldobler, and V. Mayer-Eichberger, “Extracting propositional rules from feed-forward neural networks – a new decompositional approach,” in *Proceedings of the IJCAI-07 Workshop on Neural-Symbolic Learning and Reasoning (NeSy’07)* (A. S. d. Garcez, P. Hitzler, and G. Tamburrini, eds.), vol. 230, 2007.

[120] S. Bader, “Extracting propositional rules from feed-forward neural networks by means of binary decision diagrams,” vol. 481, pp. 22–27, 2009-01.

[121] R. Andrews, J. Diederich, and A. Tickle, “Survey and critique of techniques for extracting rules from trained artificial neural networks,” *Knowledge-Based Systems*, vol. 6, pp. 373–389, 1995-12.

[122] T. Hailesilassie, “Rule extraction algorithm for deep neural networks: A review,” 2016-09.

[123] “The MNIST database of handwritten digits.” `http://yann.lecun.com/exdb/mnist/`. Accessed: 2020-02-18.

[124] W. Bolton, *Programmable logic controllers*. Newnes, 2015.