Binary Decision Diagrams Functional Specification

Jakub Dolecki jdolecki@fas.harvard.edu
Sean Murphy spmurphy@fas.harvard.edu
Michael Cioffi mcioffi@fas.harvard.edu

Brief Overview

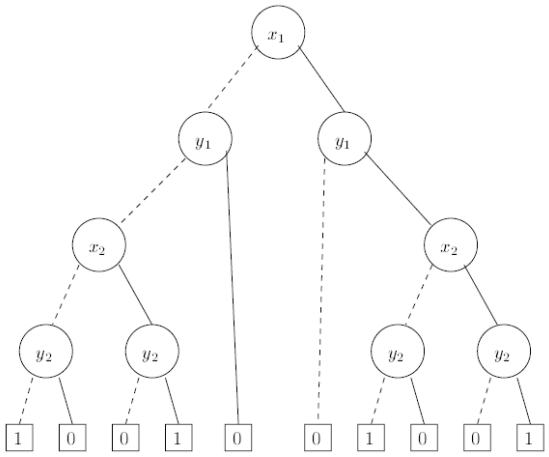
Our project aims to implement binary decision diagrams in Objective Caml (Ocaml). Binary decision diagrams (BDDs) belong to a class of data structures that represent Boolean functions. In particular, they are very useful in finding solutions to such functions as the number of variables increases because they do so efficiently, or in general O(n) time in the worst case where n is the number of variables . Complex Boolean functions arise in a variety of combinatorial problems such as circuit testing, shortest paths, etc. BDDs have been utilized to solve those types of problems for decades. Not only are BDDs useful for finding solutions to complex Boolean functions, they can also be used to visualize them when they are small. We will be utilizing both of the abovementioned characteristics of BDDs in our project when working with complex Boolean functions.
Our primary goal is to build an important and useful abstract data type by utilizing concepts learned in class. Along the way, we also seek to conceptualize a more formal definition of Boolean expressions and functions. Finally, we will apply our BDD to solve a trivial problem. We intend to use that application in our presentation at the end of class.

Feature List

Alpha features will include:

- Each future list below will be implemented in a module
- Parse a Boolean function and divide it into individual Boolean expressions or variables. In a sense, this creates a set of variables that will go into the BDD.

- From the deconstructed expressions (pictured below), construct a truth diagram as described in "An Introduction to Binary Decision Diagrams" by Andersen[1].



- From the ordinary binary decision tree, construct a Reduced Ordered BDD (ROBDD), or a "true" BDD as described in Art of Programming, Volume 4, Fascicle 1 by Knuth[2].
- A function that will apply a Boolean operator between two ROBDDs as described in "An Introduction to Binary Decision Diagrams" by Andersen[1] and return a new ROBDD.
- A function that will return all truth assignments that satisfy f(x) = 1 where f is a Boolean function as described in "An Introduction to Binary Decision Diagrams" by Andersen[1]. An extension of this function would return the number of satisfactory solutions to the constraint.
- A fold function will be useful in determining statistics about our ROBDD such as the number of nodes or depth.

Cool features

- Print function which will be used for debugging and visualization.

---

[1] http://bit.ly/grcwnZ

[2] Knuth, Donald. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. New Jersey: Addison-Wesley Professional, 2009.

- Function that takes a ROBDD and some variable assignments, such as x1 = 0 or x2 = 0 and computes a new ROBDD as described in "An Introduction to Binary Decision Diagrams" by Andersen[1]. This function will effectively reduce a ROBDD further to get rid of the assigned variables.
- Function that returns a random solution to the Boolean function as described in Art of Computer Programming by Knuth[2].
- Functions that return solutions of maximum or minimum weight if each variable is given a constant weight.

"What's Next?"

Since BDD's involve binary trees themselves, we will first try to replicate similar binary trees we have seen in lecture and homework for OCaml. Just to reiterate, OCaml will be our best option here because of two reasons: **A)** BDD's are abstract data types which have a similar structure to a binary search tree and **B)** as mentioned in the features list, efficiency with trees is an important factor. We will define signatures for our BDD's that can be operated on. In order to utilize BDD's, we need to be able to parse Boolean algebra functions and translate them into meaningful tree paths. Until the next draft, we will write a basic parser given some Boolean function, but not yet output all the possible combinations (which is essentially what the BDD will do as the final implementation). In addition, we aim to specify necessary data types and complete the signature for the entire BDD module.

Binary Decision Diagrams (Draft) Technical Specification

Jakub Dolecki jdolecki@fas.harvard.edu
Sean Murphy spmurphy@fas.harvard.edu
Michael Cioffi mcioffi@fas.harvard.edu

Modularity

The binary decision diagram (BDD) module will work much like a library. Our implementation will not rely on functors to specify the module because a node in a BDD is always of type Boolean. Thus, our module will simply hold a collection of methods available for operating on Boolean functions. Overall, we agree on an even division of our project into two parts. The first part deals with parsing an expression and building a BDD. The second part deals with all of the operations on the constructed BDDs outlined in the functional specification. Throughout our discussions, we also agreed that Jakub Dolecki will work on functions related to the former part, Michael Cioffi and Sean Murphy will handle the latter part of the project. Of course, the functions that belong to the latter part of the project will be split up evenly between Michael and Sean. The only problem that we foresee with such a split is the dependency of the latter part on the former which will require constant communication between the group members.

Interfaces

Types:

These types will be used throughout the project.

```
Type exp = And of exp * exp | Or of exp * exp | Not of exp

Type index = int

Type bdd = Node of index * bdd * bdd | Zero | One
```

Building the BDD:

```
(* Parse will look at a Boolean function and divide it into a usable type *)

Val parse = bool -> exp
```

(* Make will take a single expression and build a node *)

(* Make will ensure that the BDD is actually reduced, or that it has unique nodes only. As such, there will need to be basically a lookup table that has all of the existing nodes *)

Val make = exp -> bdd

(* Build will take a Boolean function and build an entire BDD *)

Val build = exp -> bdd