Tractability and Expressivity

Book on Measure Theory: "Measure, Integration and Real Analysis" - Sheldon Axler

When does sampling work well and when does it work poorly?

We look at it in the context of an example.

I can flip two coins:

```
x ← flip 0.000001
y ← flip 0.000001
return x ∧ y
```

Sampled: F, F, F, F, ...

The tagline is that direct sampling performs poorly for low probability estimates. You'll encounter the same problem with a mostly true probability.

Question: Isn't this okay, as its true that the probability is low?

- Yes, but another issue lies in interactions with conditioning.
- We'll modify the program to:

```
x ← flip 0.00001
y ← flip 0.00001
observe x ∧ y
return x
```

Samples
tt,
FF
tt
⊥ — rejected because it didn't match observation

How do we observe sampling?

1. Rejection Sampling

- Violation of the observation means you forget that sample and draw another one, denoting it "bottom"
- Afterwards, only consider the accepted samples
- In the program we have above, this results in lots of samples being done.
- Called the low probability of evidence problem.
- We have a worst case hardness situation, and
- Drawback of approximation sampling is dealing with approximate conditioning.

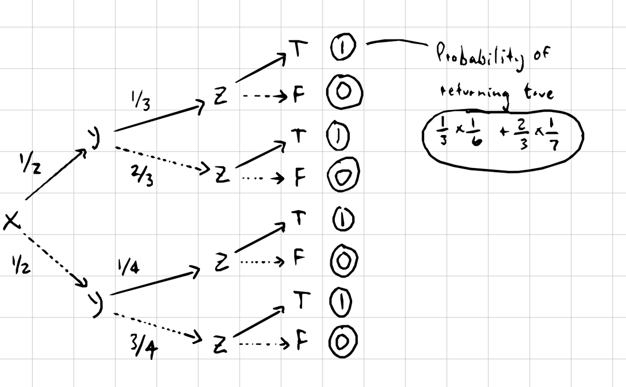
2. Search

We use a probabilistic if, which should be easy to define.

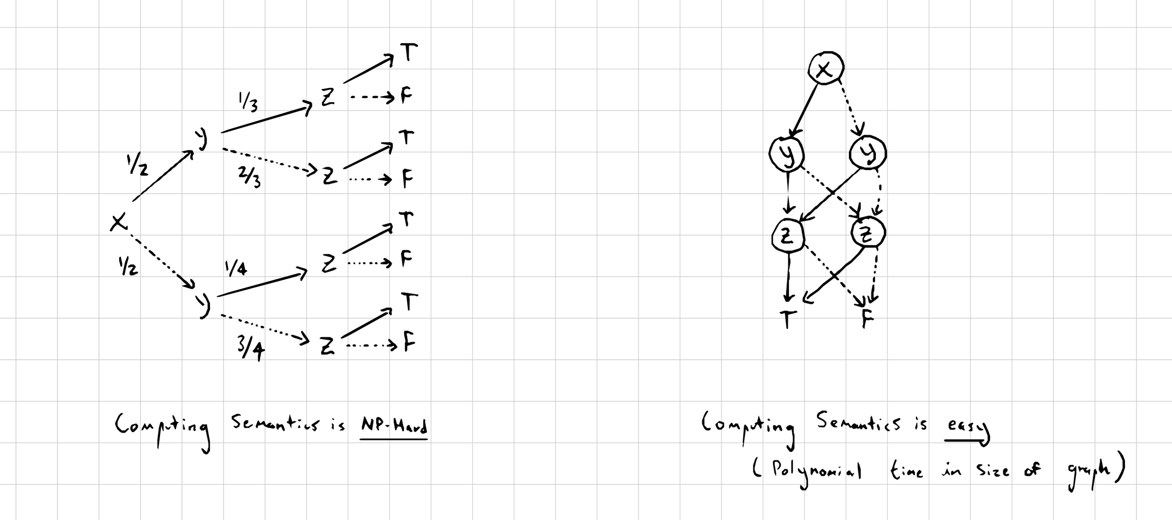
Example Program:

```
x <- flip 1/2;
y <- if x then flip 1/3 else flip 1/4;
z <- if y then flip 1/6 else flip 1/7;
return z;
```

I can write down a search tree through this program



Tree above could have been drawn more cleverly, re-using duplicate substructure.



This transformed graph above and to the right is saving us some effort. It's called a **Binary Decision Diagram.**

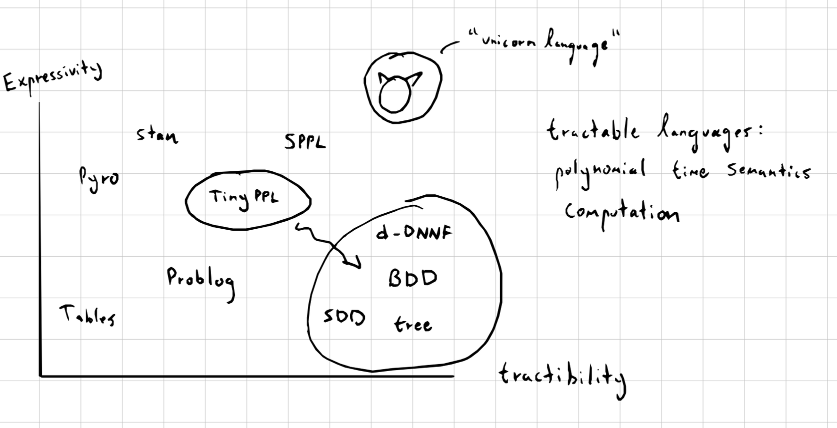
- With it, we can answer questions about probability nicely.
- How do we build this?
- Going from the larger tree to the smaller seems untenable.

Binary Decision Diagrams (BDD):

- "Only fundamental data structure" - Donald Knuth

What do I mean by tractibility and expressivity?

The language is expressive, the graph is restrictive.



Idea: Use a more tractable language as an "assembly language" for your probabilistic programs. The tractable language programs will have the same semantics.

Question: What if we were ok with losing semantics?

- Good paper idea, but we're going to talk about semantics preserving compilation

Knowledge Compilation: (Adnan Darwiche)

- His observation is that there's a relationship between hardness of propositional reasoning tasks and syntax of formulae.
- This makes sense, if we consider for instance conversion to DNF (Disjunctive Normal Form) for SAT.
- Example DNF: $(A \wedge B) \vee (A \wedge C \wedge \overline{D}) \vee \dots$
- DNF is a family of formulae for which SAT is easy to solve.

Begs the question, what kinds of structure enable efficient reasoning?

Paper: "A Knowledge Compilation Map", 2002, Adnan Darwiche.

- discusses Succintness

ore efficient to translate all programs in L2 to L1

Succintness

$\mathcal{L}_1$ is more succinct than $\mathcal{L}_2$

If it is more efficient to translate all programs in $\mathcal{L}_2$ to $\mathcal{L}_1$

- In looking back at the tractibility and expressivity graph, we say that TinyPPL is more succint than BDD.
- A Related Paper in PL: "On the Expressive Power of Programming Languages", Felleisen, 1991.
- Efficiency is polynomial in the size of the program

Question: Do we consider this individually or for all possible programs?

- It's a for-all, so, all possible programs.
- Follow-up: How can we know BDD is not more succint than TinyPPL
  - Assume that a translation from BDD to TinyPPL exists, therefore, we can compute a translation in polynomial time. This produces a contradiction

$\#p\text{ hard}$ $\text{TinyPPL} \overset{[\![\cdot]\!]}{\rightsquigarrow} \text{BDD} \longrightarrow \text{probability}$

Question: Do we consider the size of the program?

- This is also polynomial in space, because the space is restricted by the runtime complexity.

Question: Do you have an example of a very intractable language?

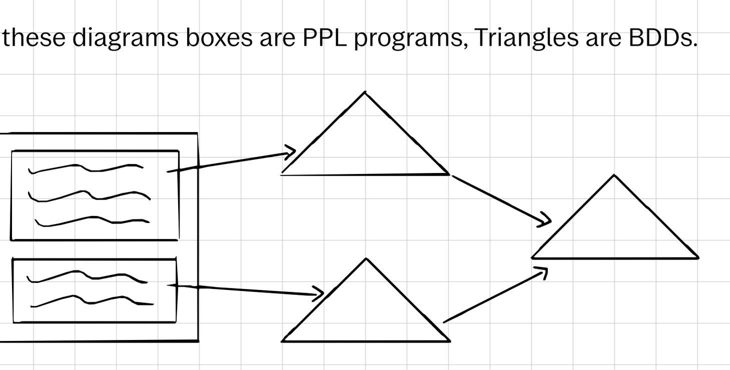
- Table language where each row is a possible world.
- Could end up with a very large table.

Compiling TinyPPL to BDD

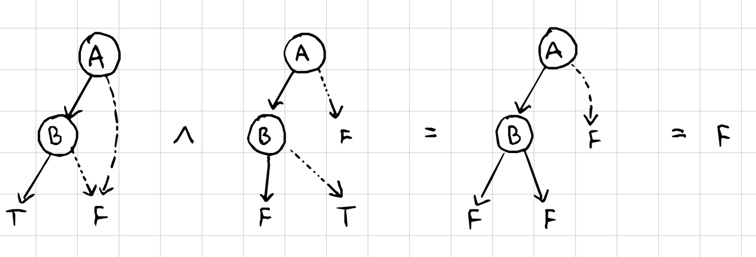
Paper: Holtzen et al, OOPSLA, 2020

We want a compositional compilation.

In these diagrams boxes are PPL programs, Triangles are BDDs.

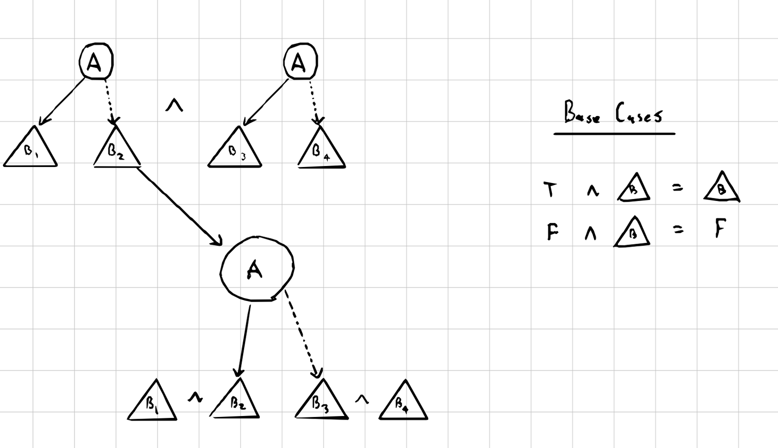


BDDs have natural compositionality.



I'm going to show a simple inductive procedure

Starting with A, conjoin the true child and the false child together.



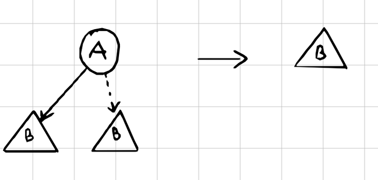
Structural Invariants of BDDs:

1. Ordering of variables is the same on all paths.

2. Reduction:

1.) No isomorphic subtrees (syntactically, no redundant subterms)

2.) No Vacuous children/nodes. This is implied by 1st part of reduction.

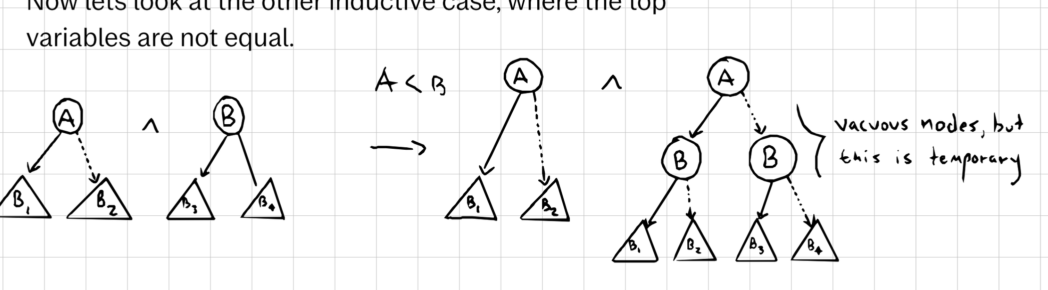


Above structural invariants imply Canonicity

(same variable ordering)

$$[\![e_1]\!] = [\![e_2]\!] \iff e_1\text{'s BDD} = e_2\text{'s BDD}$$

Now lets look at the other inductive case, where the top variables are not equal.



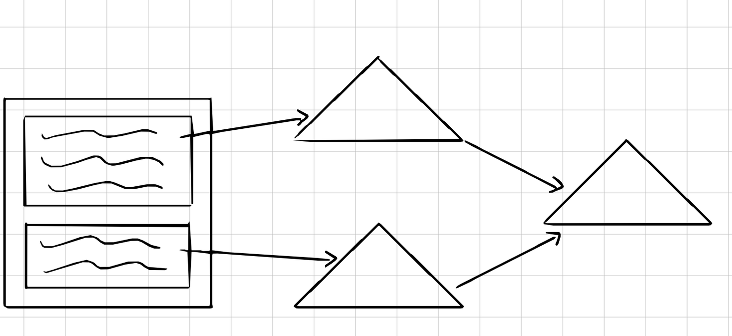
Question: What happens if we add observe?

- Won't discuss the full compilation.
- Trick is that you'll have two bdds, one representing prob of accepting, one unnormalized semantics.
- Could use ZDDs

Question: On what occurs when its a graph with two nodes pointing to the same false.

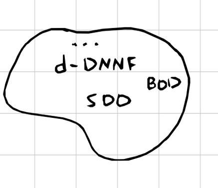
- The tree restriction is helpful here. Each node needs a true branch and a false branch.

For next time:



We take pieces of a tinyPPL program, compile them to BDDs, and then compose them.

Benefit: There are really high performance BDD libraries, similar to the situation with SAT Solvers.



Variety of approaches to compilation

Ours was bottom up compilation.

Question: Do we get a nicer theory from BDDs?

- Yea, BDDs have affine logic, sharing of subterms.
- Theory of BDDs tells you something about your inference cost
- There is some theory argument that could be made there.

We'll do a small part of the program we showed earlier

```
x ← flip 1/2;
y ← if x then flip 1/3 else flip 1/4;
return y
```

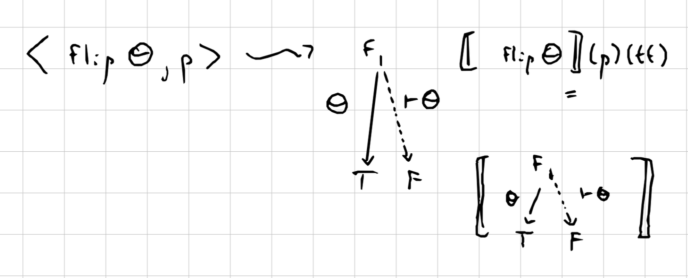
Compilation Arrow:

$\langle e, \rho \rangle \rightsquigarrow \text{BDD}$

$(1) \rightarrow \text{BDD}$

Code has an implementation of this arrow

Define this relationship inductively, starting with base case



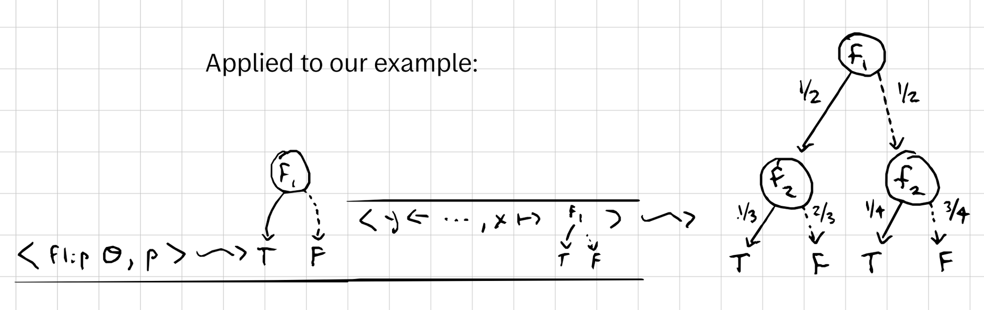
Now we do bind:

$$\frac{\langle e_1, \rho\rangle \rightsquigarrow B_1 \qquad \langle e_2, \rho[x \mapsto B_1]\rangle \rightsquigarrow B_2}{\langle x \leftarrow e_1;\ e_2, \rho\rangle \rightsquigarrow B_2}$$

Pure Terms

$$\frac{\langle p, \rho\rangle \rightsquigarrow B}{\langle \text{return } p, \rho\rangle \rightsquigarrow B}$$

Applied to our example:



This is a really effective way to do probabilistic inference. Code has examples of how to do this.

Question: What causes the complications that might make a large and inefficient program?

- Arbitrary boolean expressions in the return p,p case
- Knuth's "Art of Computer Programming" has a lot of pages on BDDs

Question: Does the order of variables actually matter?

- It is a requirement to make progress in the inductive step

Question: Are there heuristics for picking good variable orderings?

- Books of them.