Propositional Probability III: Compilation[1]

Steven Holtzen
s.holtzen@northeastern.edu

September 19, 2023

[1] CS7470 Fall 2023: Foundations of Probabilistic Programming.

- **Goal:** to evaluate terms from Prop as efficiently as possible
- Last time we saw $(\pi, (\varphi, w)) \Downarrow^e v$. We observed that it is efficient (i.e., polynomial in $|\varphi|$) if φ has specific forms: for instance, if it is a simple conjunction of variables. This time, we will see some more sophisticated evaluation strategies

1 *Memoization*

- Consider the formula $(a \vee b) \wedge (c \vee d)$ and some w.
- Let's see a sketch of a derivation tree (where we elide w and π for now):

$$\dfrac{\dfrac{(\star)\ \dfrac{\cdots}{c \vee d \Downarrow^e v_3} \qquad \texttt{false} \Downarrow^e 0}{(b \wedge (c \vee d)) \Downarrow^e v_1} \qquad \dfrac{\cdots}{(c \vee d)) \Downarrow^e v_2}\ (\star)}{((a \vee b) \wedge (c \vee d)) \Downarrow^e w(a)v_1 + w(\overline{a})v_2}$$

- **Observe**: both of the subtrees marked by $(\star)$ are *identical*. We are wasting effort by re-deriving both. We should *reuse* these derivations rather than recomputing them.
- Now we will define a new reduction rule that memoizes previously computed results
- Shape of new relation: $(\rho, \texttt{p}) \Downarrow^m (v, \rho')$, where ρ is a memoization table that maps Prop programs to probabilities:

So far we've considered *tree-like* proof-systems (also called *Gentzen sequent calculus [Gentzen, 1969]*): these are systems of deductions that describe trees, and cannot share sub-derivations. There are more powerful DAG-like proof-systems that permit the reusing of subderivations. If you are interested in learning more about the power of various proof systems, check out Beame and Pitassi [2001]

$$(\pi, \rho, (\texttt{true}, w)) \Downarrow^m (1, \rho) \qquad (\pi, \rho, (\texttt{false}, w)) \Downarrow^m (0, \rho)$$

$$(\textsc{Memo})\ \dfrac{\rho(\varphi) = v}{(\pi, \rho, (\varphi, w)) \Downarrow^e (v, \rho)}$$

$$(\textsc{Split})\ \dfrac{\begin{array}{c}\varphi \notin \rho \qquad (\pi, \rho, (\varphi[\texttt{true}/x], w)) \Downarrow^m (\rho', v_1) \\ (\pi, \rho' \cup (\varphi[\texttt{true}/x] \mapsto v_1), (\varphi[\texttt{false}/x], w)) \Downarrow^m (\rho'', v_2)\end{array}}{(x :: \pi, \rho, (\varphi, w)) \Downarrow^m (\rho'' \cup (\varphi[\texttt{false}/x] \mapsto v_2), w(x)v_1 + w(\overline{x})v_2)}$$

Analysis of memoization

- What kinds of Prop programs does our new memoized reduction scale well on?
- For the purposes of analysis, it can be useful to give a *cost-annotated* version of our rules that counts how many derivations were produced during runtime. This is fairly easy to do: we augment our relation with an integer n that accumulates the total runtime, so our new relation is $(\pi, \rho, (\varphi, w)) \Downarrow (v, \rho', n)$:

$$(\pi, \rho, (\texttt{true}, w)) \Downarrow^m (1, \rho, 1) \qquad (\pi, \rho, (\texttt{false}, w)) \Downarrow^m (0, \rho, 1)$$

$$(\textsc{Memo})\ \frac{\rho(\varphi) = v}{(\pi, \rho, (\varphi, w)) \Downarrow^e (v, \rho, 1)}$$

$$(\textsc{Split})\ \frac{\varphi \notin \rho \qquad (\pi, \rho, (\varphi[\texttt{true}/x], w)) \Downarrow^m (\rho', v_1, n_1) \qquad (\pi, \rho' \cup (\varphi[\texttt{true}/x] \mapsto v_1), (\varphi[\texttt{false}/x], w)) \Downarrow^m (\rho'', v_2, n_2)}{(x :: \pi, \rho, (\varphi, w)) \Downarrow^m (\rho'' \cup (\varphi[\texttt{false}/x] \mapsto v_2), w(x)v_1 + w(\overline{x})v_2, n_1 + n_2 + 1)}$$

- Complexity in the *elimination-width*, similar to variable elimination (won't get into that now...)

2 *Tractable runtimes*

- **Problem**: evaluating probabilities is hard for *arbitrary formulae*. What if we consider a *restricted class* of formulae for which there exists a runtime semantics that *guaranteed to be efficient in* $|\varphi|$?

  Definition 1 (Tractable runtime)**.** *Let $p \Downarrow (v, n)$ be a cost-annotated big-step reduction relation for some program p written in language L. We call $\Downarrow$ a **tractable runtime** if n is polynomial in $(|\varphi|)$ for any program $\texttt{p}$ written in L.*

- We need to define a notion of size for Prop programs. Intuitively, the size simply counts how many syntactic constructs are utilized:

$$|\texttt{true}| = 1 \qquad |\texttt{false}| = 1 \qquad \frac{|\varphi| = n}{|\neg\varphi| = 1 + n} \qquad |x| = 1$$

$$\frac{|\alpha| = n_1 \qquad |\beta| = n_2}{|\alpha \wedge \beta| = 1 + |\alpha| + |\beta|}$$

  And so on similarly for other connectives.

- Observe: neither $\Downarrow^e$ nor $\Downarrow^m$ are tractable runtimes.
  Exercise: find a family of programs that witnesses this intractability for each language.

- Can we come up with a suitable restriction to Prop that guarantees that $\Downarrow^e$ is a tractable runtime?

- **Idea**, our first tractable runtime: restrict propositional terms to be "pre-split" (or *Shannon-expanded*). Call this language Prop_S:

```
φ_S,α_S,β_S ::= true_S | false_S | (x_S∧α_S) ∨ (¬x_S∧β_S)
w_S ::= [x_S ↦ θ_x, ···]
p_S ::= (φ_S, w_S)
```

- We define a cost-annotated enumeration relation $p_S \Downarrow^e (v, n)$ in a manner identical to Prop for Prop_S

Theorem 1. *Assume $p_S \Downarrow^e (v, n)$. Then, $n \leq 1 + |p_S|$.*

Proof. By structural induction on (φ, w), inducting on φ. Base cases are easy: $(\texttt{true}, w) \Downarrow (1, 1) \leq 1 + 1$.

Now let's handle $((x \wedge \varphi_1) \vee (\neg x \wedge \varphi_2))$. Assume by induction that $\varphi[\texttt{true}/x] \Downarrow (v_1, n_1)$, $\varphi[\texttt{false}/x] \Downarrow (v_2, n_2)$, and $n_1 \leq 1 + |\varphi[\texttt{true}/x]|$ and $n_2 \leq 1 + |\varphi[\texttt{false}/x]|$.

Then, by applying the (Split) rule:

$$\frac{\varphi[\texttt{true}/x] \Downarrow (v_1, n_1) \qquad \varphi[\texttt{false}/x] \Downarrow (v_2, n_2)}{((x \wedge \varphi_1) \vee (\neg x \wedge \varphi_2)) \Downarrow^e (w(x)v_1 + w(\overline{x})v_2, 1 + n_1 + n_2)}$$

Now we are done if we can conclude that $n_1 + n_2 \leq |\varphi|$. Continuing:

$$\begin{aligned}
n_1 + n_2 &\leq 1 + |\varphi[\texttt{true}/x]| + 1 + |\varphi[\texttt{false}/x]| && \text{by I.H.} && (1)\\
&\leq 1 + |\varphi_1[\texttt{true}/x]| + 1 + |\varphi_2[\texttt{false}/x]| && (\star) && (2)\\
&\leq 1 + |\varphi_1| + 1 + |\varphi_2| && (\star\star) && (3)\\
&\leq |(x \wedge \varphi_1) \vee (\neg x \wedge \varphi_2)| && (\dagger) && (4)
\end{aligned}$$

where $(\star)$ follows from the definition of substitution, i.e. $\varphi[\texttt{true}/x] = (x \wedge \varphi_1) \vee (\neg x \wedge \varphi_2)[\texttt{true}/x] = \varphi_1[\texttt{true}/x]$;

$(\star\star)$ follows from a simple lemma: substitution can only reduce the size of the formula, i.e. $|\varphi[v/x]| \leq |\varphi|$ for any variable x and value v.

$(\dagger)$ follows from the definition of $|\cdot|$. □

Semantics-preserving translations

- **Idea**: we can give a semantics-preserving translation from an intractable language into a tractable one

- Why would we want to do that?

- Abstraction: just like we don't want all programmers to program in assembly, it's useful to identify good target languages
- Modular reasoning: We can separate our efficiency argument into 2 stages: proving the target efficient, and proving the translation efficient (or characterizing the efficiency of the translation)

- Compilation from Prop to Prop_S: define a relation $\rightsquigarrow$ that translates each term in each language while preserving semantics:

Figure 1: Compilation from Prop to Prop_S.

$$\texttt{true} \rightsquigarrow \texttt{true}_S \qquad \texttt{false} \rightsquigarrow \texttt{false}_S \qquad x \rightsquigarrow x_S$$

$$\frac{x \text{ free in } \varphi \qquad \varphi[\texttt{true}/x] \rightsquigarrow \varphi_1 \qquad \varphi[\texttt{false}/x] \rightsquigarrow \varphi_2}{\varphi \rightsquigarrow (x_S \wedge \varphi_1) \vee (\neg x_S \wedge \varphi_2)}$$

- We will want to prove this compilation correct by proving that it preserves denotation:

Definition 2 (Semantics preserving compilation)**.** *Let L_1 and L_2 be languages with semantic interpretations $[\![\cdot]\!]$ with the same semantic domain. A compilation $\rightsquigarrow$ from language L_1 to language L_2 is **semantics-preserving** if, for any program p_1 written in L_1 where $p_1 \rightsquigarrow p_2$, it is the case that $[\![p_1]\!] = [\![p_2]\!]$.*

Theorem 2. *The compilation rules in Figure 1 are semantics-preserving.*

- In a seminal paper Felleisen [1990] distinguished between different kinds of programming languages based on their supported syntactic features. He compared languages on the basis of *expressivity*: whether or not it was possible to express all programs in one language in the other.

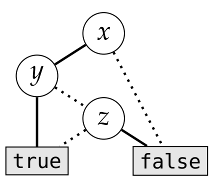
- Here we consider a refinement of this notion to *efficient expressiveness*, which captures whether or not an *efficient* (i.e., polynomial-time) translation exists between two languages:

Definition 3 (Efficient expressiveness)**.** *A program p_1 written in language L_1 is **efficiently expressible** as a program p_2 written in language L_2 if there exists a polynomial-time semantics-preserving translation $p_1 \rightsquigarrow p_2$. We say language L_1 is **efficiently expressible** as L_2, written $L_1 \sqsubseteq L_2$, if all programs in L_1 are efficiently expressible as programs in L_2.*

- Observe: Prop_S is efficiently expressible as Prop, but not vice versa

3 *Binary decision diagrams*

- **Problem**: $\textsc{Prop}_S$ is not very expressive
- **Goal:** More expressive tractable languages
- What can we improve about $\textsc{Prop}_S$? Observe that there may be *repetitious sub-syntactic terms*, and we'd like to exploit those, just like we observed in our memoized reduction scheme
- How can we represent repetitious subterms syntactically? By changing our syntax to permit *directed acyclic graphs*!
- Syntax of **binary decision diagram** (BDD): a rooted binary directed acyclic graph with two kinds of nodes:
  - *Choice nodes*, written $\texttt{Ite}(x, l, h)$, where x is a propositional variable (called the top variable), l is an edge called the *low edge*, and h is an edge called the *high edge*
  - *Terminal nodes*, which are labeled `true` or `false`.
- The syntax of BDD is given as graphical notation, for example:

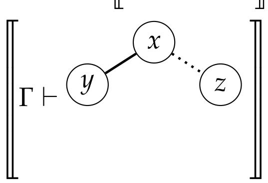


Solid edges are high edges, dotted edges are low edges

- Semantics of BDD:

$$\left\llbracket \Gamma \vdash \boxed{\texttt{true}} \right\rrbracket = \llbracket \Gamma \rrbracket$$

$$\left\llbracket \Gamma \vdash \boxed{\texttt{false}} \right\rrbracket = \emptyset$$



$$= \{I \in \llbracket y \rrbracket \mid I(x) = \texttt{true}\} \cup \{I \in \llbracket z \rrbracket \mid I(x) = \texttt{false}\}$$

References

Paul Beame and Toniann Pitassi. Propositional proof complexity: Past, present. *Future*, pages 42–70, 2001.

Matthias Felleisen. On the expressive power of programming languages. In *European Symposium on Programming*, pages 134–151. Springer, 1990.

Gerhard Gentzen. Investigations into logical deduction. *The collected papers of Gerhard Gentzen*, pages 68–131, 1969.