# Propositional Probability III: Compilation[1]

*Steven Holtzen*
*s.holtzen@northeastern.edu*

*September 19, 2023*

- **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 $\varphi$ 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 $\pi$ for now):

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

- **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: $(\rho, \mathsf{p}) \Downarrow^m (v, \rho')$, where $\rho$ is a memoization table that maps Prop programs to probabilities:

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

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

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

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]

*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 $(\pi, \rho, (\varphi, w)) \Downarrow (v, \rho', n)$:

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

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

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

- 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 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:

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

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

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$_S$:

```
1  φ_S, α_S, β_S ::= true_S | false_S | (x_S∧α_S) ∨ (¬x_S∧β_S)
2  w_S ::= [x_S ↦ θ_x, ···]
3  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 $(\varphi, w)$, inducting on $\varphi$. Base cases are easy: $(\text{true}, w) \Downarrow (1, 1)$ and $1 \leq 1 + 1$.

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

Then, by applying the (SPLIT) rule:

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

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

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

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

$(\star\star)$ 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:

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

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

Figure 1: Compilation from PROP to PROP$_S$.

- 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**: $\mathrm{P}\textsc{rop}_S$ is not very expressive

- **Goal:** More expressive tractable languages

- What can we improve about $\mathrm{P}\textsc{rop}_S$? 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 $\mathtt{Ite}(x, l, h)$, 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 $\mathtt{true}$ or $\mathtt{false}$.

- The syntax of Bdd is given as graphical notation, for example:
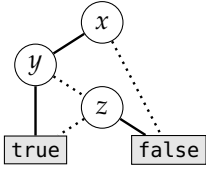
  

  Solid edges are high edges, dotted edges are low edges

- Semantics of Bdd:

$$\left[\!\!\left[\, \Gamma \vdash \boxed{\mathtt{true}} \,\right]\!\!\right] = [\![\Gamma]\!]$$

$$\left[\!\!\left[\, \Gamma \vdash \boxed{\mathtt{false}} \,\right]\!\!\right] = \varnothing$$



$$= \{I \in [\![y]\!] \mid I(x) = \mathtt{true}\} \cup \{I \in [\![z]\!] \mid I(x) = \mathtt{false}\}$$

## 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.