Towards Symbolic Execution for Probability and Nondeterminism

Jack Czenszak, John M. Li, Steven Holtzen {czenszak.j,li.john,s.holtzen}@northeastern.edu

Northeastern University Khoury College of Computer Sciences

Background

Symbolic executors use *symbolic variables* in order to compactly represent nondeterministic computations:

```
let x = sym () ;;   x = {a: true, ¬a: false}
                    σ = {(a, bool)}

let y = sym () ;;   y = {b: true, ¬b: false}
                    σ = {(a, bool), (b, bool)}

if x then
  if y then
    [1; 2]
  else
    [3]
else
→ [4] ;;

> {a ∧ b: [1; 2], a ∧ ¬b: [3], ¬a: [4]}
```

In concurrent work, we[1] have adapted the symbolic executor Rosette to perform probabilistic inference:

```
# let x = sym () ;;
let x = flip (7/10) ;;  x = {a: true, ¬a: false}      ω = {a: 7/10}
                        σ = {(a, bool)}
# let y = sym () ;;
let y = flip (2/5) ;;   y = {b: true, ¬b: false}      ω = {a: 7/10, b: 2/5}
                        σ = {(a, bool), (b, bool)}
if x then
  if y then
    [1; 2]
  else
    [3]
else
  [4] ;;

> {a ∧ b: [1; 2], a ∧ ¬b: [3], ¬a: [4]}
  {[1; 2]: 7/25, [3]: 21/50, [4]: 3/10} ← Result of WMC
```

This adaptation was hard to formally justify: though probabilistic Rosette's implementation reuses large parts of Rosette's code base, its correctness proof could not similarly reuse Rosette's correctness proof, and even required nontrivial extensions to existing models of Rosette.

This work aims for a *crisp mathematical justification* for when/why Rosette can be repurposed in this way.

[1] With Cameron Moy (moy.cam@northeastern.edu)

Towards a unifying mathematical model

We present a model of symbolic execution encompassing both nondeterminism and probability. It yields a unified correctness proof for both (a fragment of) Rosette and its probabilistic adaptation, and generalizes beyond probability and nondeterminism to any effect that is *affine commutative:*

```
let _ = M in    affine              let x = M in    commutative    let y = N in
N                 ≡       N         let y = N in         ≡         let x = M in
                                    O                              O
```

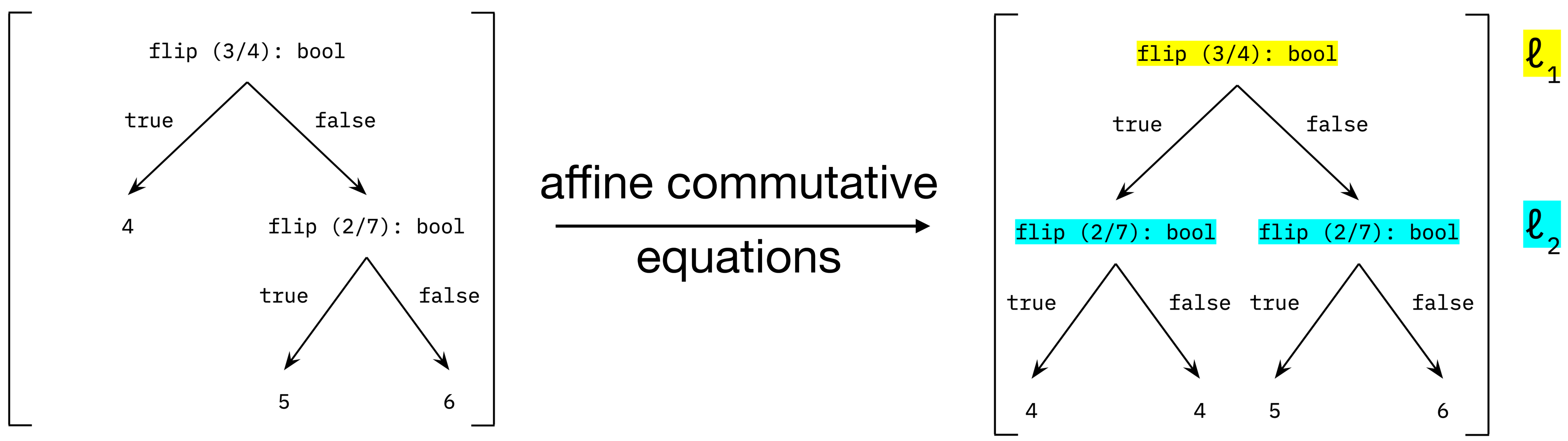
Let T be an affine commutative monad on Set presented by an algebraic theory $(\Sigma, \mathbb{T})$, giving concrete semantics to programs. We define a category W and a monad $\hat{T}$ on $[W; \mathsf{Set}]$ that gives symbolic semantics. The category W is a variant of FinInj and the monad $\hat{T}$ is a variant of Stark's name generation monad on $[\mathsf{FinInj}; \mathsf{Set}]$:

$$W = (\mathsf{FinInj} \hookrightarrow \mathsf{Set}) \downarrow (\mathsf{Set} \xleftarrow{\Sigma} 1) \qquad \hat{T}F(L, f : L \to \Sigma) = \int^{(L', f' : L' \to \Sigma)} F(L + L', [f, f'])$$

Our main result says that generating fresh symbolic names using the symbolic monad $\hat{T}$ correctly simulates the concrete monad T, and that every concrete computation is symbolically representable.

Theorem 1. There exists a "lifting" functor $L : \mathsf{Set} \to [W; \mathsf{Set}]$ and a "concretization" map $\chi : \hat{T}L \to LT$ that is surjective and commutes with operations of Σ and strong monad operations.

Key idea of the proof: every $t : TA$ can be brought into a normal form.



(The labels ℓ_1 and ℓ_2 correspond to variables allocated by the symbolic semantics $\hat{T}$.)

In future work, we plan to consider other affine commutative effects (e.g., weights) and substantiate our denotational model with a concrete programming language and implementation.