EasyChair Preprint

№ 13257

Looking Back: a Probabilistic Inverse Perspective on Test Generation

Joachim Kristensen, Tobias Reinhard and
Michael Kirkedal Thomsen

EasyChair preprints are intended for rapid
dissemination of research results and are
integrated with the rest of EasyChair.

May 12, 2024

Looking Back: A Probabilistic Inverse Perspective on Test Generation

Joachim T. Kristensen[1], Tobias Reinhard[2,3*†], and Michael K. Thomsen[1,4]

[1] Department of Informatics, University of Oslo, Oslo, Norway
{joachkr,michakt}@ifi.uio.no
[2] KU Leuven, Leuven, Belgium
[3] Technical University of Darmstadt, Darmstadt, Germany
[4] Department of Computer Science, University of Copenhagen, Copenhagen, Denmark

1 Introduction

Software validation is hard, among others things because of the sheer size of the input space [7, 18]. Reversible computation has shown promises to mitigate some difficulties in software debugging [3, 6], but has not been applied to the wider area of software validation. To alleviate this, we propose to relax reversible computing to a combination of probabilistic [4, 14] and inverse computation [2, 9, 12]. This will create a new model that is a great candidate for mitigating the difficulties of software validation.

Imagine being able to find the cause of any error by simply calling a program's inverse on it. A significant limitation to this naïve approach is that the program needs to be injective; unfortunately, most programs do not have this property. To overcome this limitation, we characterise the inverse of a program in terms of a probabilistic program as follows:

Let $f : A \to B$ be a function, and consider two inputs $a_0 \neq a_1 \in A$ such that $f(a_0) = f(a_1)$. There exists no function $g : B \to A$ such that $g \circ f = id_A$. Instead, we consider the function $\texttt{INVERT}(f) : B \to \delta(A)$ [1] which maps each output $f(a)$ to a probability distribution over the corresponding possible inputs $\delta : A \to \texttt{Prob}$. We call INVERT(f) the *probabilistic invers* of f. Consider any output $b \in B$. The probability distribution $\delta_b = \texttt{INVERT}(f)(b)$ maps each possible input $a \in A$ to the likelihood $\delta_b(a) \in \texttt{Prob}$ of "b originating from a". Note that this function is guaranteed to exist, which cannot be said for classic inverse functions. Moreover, the latter are generally partial functions, which complicates reasoning about them. Meanwhile our probabilistic inverses are guaranteed to be total.

In conventional reversible programming languages, programs are guaranteed to be (globally) invertible by enforcing a strict syntactic discipline [8,17,19]. Programs may only be comprised of locally isomorphic parts and combinations are restricted to preserve their isomorphic properties. These restrictions do not apply to the programs we consider. Moreover, we conjecture that our probabilistic inverses can be used to reason about the quality of test generators such as `QuickCheck` [1, 10], or perhaps even to derive such generators.

2 Test Case Generation

Generating input for `QuickCheck` properties is hard for at least two reasons: (i) The search space is huge. (ii) The counterexamples usually refer to edge cases that correspond to low-

*This research is partially funded by the Research Fund KU Leuven, and by the Cybersecurity Research Program Flanders.
†This research is partially funded by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297.

[1] We abbreviate the space of probability distribution over A by $\delta(A) := (A \to \texttt{Prob})$.

probability events. Knowing the probability distribution allows to (i) trim the search space and (ii) identify edge cases by favouring low-probability input. To illustrate these challenges, let us consider the following expression evaluation function (cf. Appendix A for full definition):

```
eval env (Val v      ) = return v
eval env (Var x      ) = env x
eval env (Let x e0 e1) = eval (extend x (eval env e0) env) e1
```

Suppose further, that our goal is to generate expressions for testing type preservation. That is, there is another function `typeOf`, and we want to check the property:

```
prop_type_preservation e = (typeOf e == typeOf (Val (eval emptyEnv e)))
```

Randomly generating test expressions is unlikely to yield useful results. Most expressions are not closed and many do not contain any bindings at all. Considering any such expressions will not help us in testing our type preservation property. In contrast, the expressions we are interested in only make up a small fraction of the entire input space: Closed expressions with a fair number of bindings. We capture this limitation with the following predicate:

```
interesting e = closed e && numberOfVars e `elem` normalDistribution (6, 3)
```

Interesting expressions can have an arbitrary many variables. We specify the number of bindings as probability distribution: A normal distribution of 6 variables and standard derivation of 3.

3 Probability Type System

This work aims to characterise probabilistic inverses $\texttt{INVERT}(f) : B \to \delta(A)$ of non-injective programs $f : A \to B$. The main challenge in addressing this goal is to extract the inverse probability distribution $\delta \in \delta(A)$. We propose to solve this via a type system that augments types τ by distributions δ. Consequently, our typing relation has the form $e : (\tau, \delta)$. There are four main cases to consider: (i) values of base types (i.e. non-function types), (ii) built-in injective operations, (iii) non-injective operations and (iv) control flow.

Base Values and Injective Operations: Expressions of any base type do not take any input. To avoid special treatment, we extract the trivial distribution $\delta : () \mapsto 1$. Treating built-in injective operations $op : A \to B$ is similarly straightforward. For each $b \in B$, we can extract the distribution $\delta_b : A \to \texttt{Prob}$ by setting $\delta_b(a) = 1$ if $op(a) = b$ and $\delta_b(\_) = 0$ otherwise.

Non-Injective Operations: Extracting distributions for expressions involving non-injective operations like $+$ is more challenging. We propose to handle such expressions by following the structure of the AST and combining the subexpressions' distributions. The biggest challenge is finding a sound distribution composition $\delta_0 \oplus \delta_1$, satisfying the typing judgement

$$\texttt{+}(\tau, \delta) : \frac{\Gamma \vdash e_0 : (\tau, \delta_0) \quad \Gamma \vdash e_1 : (\tau, \delta_1)}{\Gamma \vdash e_0 + e_1 : (\tau, \delta_0 \oplus \delta_1)} \; .$$

Control Flow and Probability Bounds: We can treat branching control flow similarly to non-injective operations. The main challenge is to extract distributions for each branch and to combine them. A branch's probability is proportional to the fraction of inputs for which an execution follows said branch. To avoid computability issues, we resort to over-approximations, i.e., extracting upper bounds on the probability of each branch. This requires us to relax our notion of probability distributions and accept that the sum of all branch bounds exceeds 1. Yet, it is important to note that this will still allow us to reason about the quality of test generators.

Related Work: Previous work has defined a semantics for probabilistic programming with higher-order functions [15]. In [5], this work was extended to allow a structured way to formulate statistics with the possibility to work outside the standard measure-theoretic formalization of probability theory. This work has so far concluded in the paradigm of exact conditioning for observations in probabilistic programs [16]. A method for automatically deriving Monte Carlo samplers from probabilistic programs [13] has applied automatic differentiation and transformation inversion, while [11] have relaxed the usual PPL design constraint to achieve a richer language model.

References

[1] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279. ACM, 2000.

[2] Edsger W. Dijkstra. *Program Inversion*. Springer, 1979.

[3] Jakob Engblom. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, pages 1–6. IEEE, 2012.

[4] Mor Harchol-Balter. *Introduction to probability for computing*. Cambridge University Press, 2023.

[5] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. A convenient category for higher-order probability theory. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, 2017.

[6] James Hoey, Ivan Lanese, Naoki Nishida, Irek Ulidowski, and Germán Vidal. A case study for reversible computing: Reversible debugging of concurrent programs. *Reversible Computation: Extending Horizons of Computing: Selected Results of the COST Action IC1405 12*, pages 108–127, 2020.

[7] John Hughes. Experiences with QuickCheck: Testing the hard stuff and staying sane. In *A List of Successes That Can Change the World*, pages 169–186. Springer International Publishing, 2016.

[8] Petur Andrias Højgaard Jacobsen, Robin Kaarsgaard, and Michael Kirkedal Thomsen. CoreFun: A typed functional reversible core language. In Jarkko Kari and Irek Ulidowski, editors, *Reversible Computation*, pages 304–321. Springer International Publishing, 2018.

[9] Joachim Tilsted Kristensen, Robin Kaarsgaard, and Michael Kirkedal Thomsen. Jeopardy: An invertible functional programming language, 2022.

[10] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. Beginner's luck: A language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 114–129. ACM, 2017.

[11] Alexander K. Lew, Matin Ghavamizadeh, Martin C. Rinard, and Vikash K. Mansinghka. Probabilistic programming with stochastic probabilities. *Proc. ACM Program. Lang.*, 7(PLDI), 2023.

[12] Kazutaka Matsuda and Meng Wang. Sparcl: a language for partially-invertible computation. *Proc. ACM Program. Lang.*, 4(ICFP), 2020.

[13] David A. Roberts, Marcus Gallagher, and Thomas Taimre. Reversible jump probabilistic programming. In Kamalika Chaudhuri and Masashi Sugiyama, editors, *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics*, volume 89 of *Proceedings of Machine Learning Research*, pages 634–643. PMLR, 16–18 Apr 2019.

[14] Sheldon M Ross. *Introduction to probability models*. Academic press, 2014.

[15] Sam Staton, Hongseok Yang, Frank Wood, Chris Heunen, and Ohad Kammar. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, page 525–534, New York, NY, USA, 2016. Association for Computing Machinery.

[16] Dario Stein and Sam Staton. Probabilistic programming with exact conditions. *J. ACM*, 71(1), feb 2024.

[17] Michael Kirkedal Thomsen and Holger Bock Axelsen. Interpretation and programming of the reversible functional language. In *Symposium on the Implementation and Application of Functional Programming Languages*, IFL '15, pages 8:1–8:13. ACM, 2016.

[18] J.A. Whittaker. What is software testing? And why is it so hard? *IEEE Software*, 17(1):70–79, 2000.

[19] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *Partial Evaluation and Program Manipulation. PEPM '07*, pages 144–153. ACM, 2007.

A Full Type Preservation Example

```
data Expr
  = Val Value
  | Var Name
  | Let Name Expr Expr

data Value
  = VInt  Integer
  | VBool Bool

type Name  = String
type Error = String
type Env   = Name -> Either Error Value

emptyEnv :: Env
emptyEnv x = Left $ "unbound variable " ++ x

extend :: Eq a => a -> b -> ((a -> b) -> (a -> b))
extend x v env y = if x == y then v else env y

eval :: Env -> Expr -> Either Error Value
eval _   (Val v      ) = return v
eval env (Var x      ) = env x
eval env (Let x e0 e1) = eval (extend x (eval env e0) env) e1

closed :: Expr -> Bool
closed e = (freeVars e == [])

freeVars :: Expr -> [Name]
freeVars (Val _      ) = [ ]
freeVars (Var x      ) = [x]
freeVars (Let x e0 e1) = freeVars e0 ++ filter (/= x) (freeVars e1)

numberOfVars :: Expr -> Int
numberOfVars (Val _      ) = 0
numberOfVars (Var _      ) = 1
numberOfVars (Let _ e0 e1) = numberOfVars e0 + numberOfVars e1
```

```haskell
data Type
  = TInt
  | TBool

typeOf :: Expr -> Type
typeOf = typeOf' (const undefined)
  where
    typeOf' _ (Val (VInt  _)) = TInt
    typeOf' _ (Val (VBool _)) = TBool
    typeOf' r (Var        x ) = r x
    typeOf' r (Let x e0 e1)   = typeOf' (extend x (typeOf' r e0) r) e1

prop_type_preservation :: Expr -> Bool
prop_type_preservation e = (typeOf e == typeOf (Val (eval emptyEnv e)))

-- The goal of this research
instance Arbitrary Expr where
  arbitrary = probabilisticInverse interesting True

interesting :: Expr -> Bool
interesting e = closed e && numberOfVars e `elem` normalDistribution (6, 3)
```