Verified Inverse Function Search for Normalizing Flows (DRAFT)

David Richter
Technical University of Darmstadt, Germany

Timon Böhler
Technical University of Darmstadt, Germany

Benedict Smit
Technische Universität Darmstadt

Pascal Weisenburger
University of St. Gallen, Switzerland

Mira Mezini
Technical University of Damstadt, Germany
The Hessian Center for Artificial Intelligence (hessian.AI), Germany

Abstract

Libraries for probabilistic modeling typically rely on manually defined forward and inverse transformations, a tedious process that has spurred interest in automated inversion. However, existing automated solutions to program inversion are often ill-suited for modern machine learning. Program inversion approaches typically mandate strict local invertibility and more flexible approaches lack mechanized formalizations and proofs of correctness. On the other hand, research on exact probabilistic inference has focused on expressive features like recursion that are not often used in ML architectures, as they map poorly to GPU accelerators.

In this paper, we propose a verified algorithm for inversion by semi-inversion. Rather than requiring strict local invertibility, our approach treats inversion as a search problem: it seeks a computational path that recovers input variables from outputs by composing partial inverses. We fully mechanize the algorithm and its soundness proof in Lean, leveraging intrinsically scoped and typed syntax to ensure correctness. To our knowledge, this is the first mechanized soundness proof for a semi-inversion algorithm capable of handling non- and partially invertible primitives.

As a case study, we consider Normalizing Flows, generative models that construct complex distributions via sequences of simple transformations, which contain invertible, non-invertible and partially invertible operations. We show that our algorithm automatically synthesizes inverses for standard flows, including additive coupling, residual, and autoregressive layers effectively handling the non-invertible arithmetic and data dependencies that defeat traditional inversion techniques. Existing automated solutions from the probabilistic programming world, while featuring advanced constructs fail to invert even simple flow architectures that use simple expressions and do not feature dynamic control flows.

2012 ACM Subject Classification Author: Please fill in 1 or more \ccsdesc macro

Keywords and phrases Author: Please fill in \keywords macro

Digital Object Identifier 10.4230/LIPIcs...

Funding *Timon Böhler*: LOEWE/4a//519/05/00.002(0013)/95
Mira Mezini: LOEWE/4a//519/05/00.002(0013)/95; HMWK cluster project *The Third Wave of Artificial Intelligence* (3AI).

1 Introduction

In libraries for programming with probabilities in the context of machine learning architectures [23, 26, 25, 27], probability transformations are often defined together with their inverse

© Author: Please provide a copyright holder;
licensed under Creative Commons License CC-BY 4.0

transformations to calculate both transformed values and transformed densities. Typically, both directions are written by hand, yet manually implementing inverses is tedious and recently interest in automatically calculating the inverse functions is growing [24, 20].

However, existing automated solutions to program inversion are often ill-suited for modern deep probabilistic learning. One class of works on exact probabilistic inference [6, 7] and program inversion [16] has focused on issues for expressivity such as higher-order functions and recursion. Specifically in the machine learning domain, however, these features have limited applicability as they all allow for dynamic control flow which does not align well with the computational model of GPUs. Most reversible programming languages [29, 8], on the other hand, offer only fully invertible operations and by doing so guarantee whole program invertibility. However, requiring *invertibility* for every operation is too restrictive as programs can be invertible even if some sub-expressions are not, which has motivated relaxed forms of invertibility [9]. Although a few works have explored inversion algorithms that relax the requirement of full reversibility [12, 16, 13, 9], no mechanized soundness argument has yet been established for them. In particular, beyond intuitive justification, there is currently no mechanized formalization and proofs for program inversion by semi-inverses.

In this paper, we address the above gaps in the state of the art. Rather than increasing the expressivity of the underlying language, our work focuses on automatically inverting programs that contain non-invertible functions and partially-invertible functions (like addition, whose inversion requires one argument to be known) – this form of program inversion is best described as a search. Notably, we formally define an inversion algorithm based on semi-inverses and provide a mechanized proof of its soundness.

Whereas the (full) *inverse* of a function with multiple arguments and multiple outputs reconstructs all inputs *from all outputs*, a *semi-inverse* is any function that takes *a subset of the inputs and outputs* and reconstructs the remaining ones. Semi-inverses can often be constructed even when a full inverse does not exist by analyzing the functional dependencies among variables. For example, consider the addition expression $z = x + y$. It is impossible to recover both x and y from z alone, so addition has no full inverse. However, once x is known, one can recover y as $z - x$. In fact, $z = x + y$ has three semi-inverses: $x = z - y$, $y = z - x$, and $z = x + y$ itself.[1]

Essentially, we propose an algorithm invert as a partial function, which takes a program f and returns either a program g that is a left inverse of f, or nothing,[2] if it cannot invert f:

$$\forall f, g.\ \mathsf{invert}\, f = \mathsf{some}\, g \implies \forall x.\, g\,(f\,x) = x$$

Given that invert is partial, this implication captures only soundness, not completeness. Indeed, many useful programs have no inverse, so completeness cannot be achieved for all programs. However, while the inversion process itself is partial, the programs it produces are *total*. Soundness therefore means that whenever an inverse program g can be generated, we can statically guarantee – without running g – that it computes correct inverses for all inputs. More precisely, for any program written in a language satisfying the conditions outlined in Section 3, all inverse programs automatically derived by our algorithm are provably correct, i.e., compute correct inverses for all inputs.

The central challenge in establishing the mechanized soundness proof lies in relating the constructed inverse program g to the original one f in a way that allows us to conclude

[1] Considering every function to be a semi-inverse of itself streamlines the definition of semi-inverses, and reduces special cases.
[2] $\mathsf{some}\, g$ is a non-empty optional value.

that g is indeed an inverse of f. Our key insight is that, although g may use different operations, every (intermediate) value computed in its evaluation trace also appears in the evaluation trace of f, though possibly in a different order. By working in A-normal form (ANF) [5], every sub-expression of the program is assigned a unique variable name, then we formalize this relationship by defining a mapping from variables in g to variables in f, requiring that corresponding variables evaluate to the same value. We refer to this mapping as *concordance*.[3]

The remaining question is how to generate an inverse program with a concordance to the original program. For this, we define program inversion as an algorithm structured into four steps: (1) Given the program to invert f in A-normal form, we *desequentialize* it into a set of (unordered) primitive equations, one per variable, while asserting that the generated equations are valid, i.e., the left and the right hand sides evaluate to the same value in the original program. (2) We *enrich* the set of equations with their semi-inverses, thereby proving that the additional equalities are still valid in the original program. (3) We *sequentialize* the set of equations resulting from enrichment into the inverse program g, by incrementally selecting the equations to add to g in an order that ensures that the program being constructed remains well-scoped and thus well-defined, while simultaneously building the concordance that maps the variables of g to variables in f. (4) Now, if the concordance maps g's output variables to the original inputs of f, then g indeed computes an inverse of f. This observation leads to an elegant proof using intrinsically scoped and intrinsically typed syntax.

We prove our program inversion procedure sound, i.e., if our inversion procedure returns a program, this program is guaranteed to be the inverse of the input program. While our procedure is not complete, we demonstrate that it can invert nontrivial example programs that are paradigmatic for the use case of 'a program that contains noninvertible operations but is still invertible', namely several types of layers used in normalizing flows [17]. This is in particular interesting, as we show that even though existing exact probabilistic inference tools can handle advanced features such as higher-order functions or recursion (which may not be so relevant in machine learning) they fail to invert even simple normalizing flows.

Contributions. In summary, we make the following contributions:

- We provide the first formalization for a program inversion algorithm based on semi-inversion, starting with an overview in Section 3.
- We introduce a proof method, fit for mechanized verification, to show that the inverted program computes (a subset of) the values of the original program based on the novel concept of a *concordance*, which maps variables of the inverted program to the variables in the original program that contain the same value (Section 4).
- We formalize and implement an algorithm that constructs an inverted program along with a concordance and mechanize the algorithm and its correctness proof in the Lean theorem prover (Section 5).
- We demonstrate the applicability of the algorithm to nontrivial examples by instantiating it some invertible programs stemming from the literature on normalizing flows (Section 6).

To the submission, we attached a supplement.zip file containing our mechanization and the normalizing flow case study.

[3] Inspired from linguistics, where a *concordance* is form of an index in a book, which maps words to their occurrences within the text.

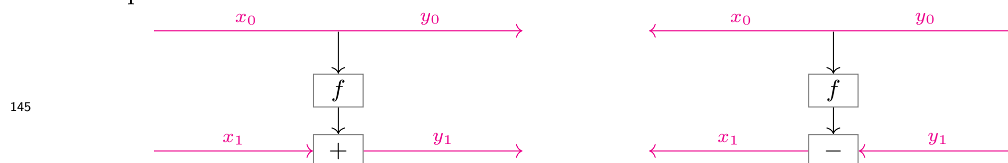
2 Program Inversion By Example

We illustrate our inversion algorithm on a normalizing flow, demonstrating how it handles non-invertible operations and non-sequential dataflow dependencies

In particular, we are interested in the additive coupling flow. Given any function f (invertible or not), an additive coupling flow is the function taking x_0 and x_1 to y_0 and y_1 via the following equation:

$$y_0 = x_0; \quad y_1 = f(x_0) + x_1$$

There are many practical constructions that contain non-invertible operations – additive flows are just one example of such constructions. At first, it might seem difficult to invert the above program, as f is not necessarily invertible and addition is a binary function which has no inverse. Yet, we actually do not need to invert f in order to invert the flow. This becomes easy to see once we visualize the dataflow dependencies below, where we highlighted the paths between outputs and inputs, demonstrating that inversion is only needed along those paths:



The flow maps x_0 to y_0 and $f(x_0) + x_1$ to y_1 (left diagram). To create an inverse, the incoming and outgoing data flow directions are reversed (right diagram). To adapt to the changed direction of data flow, *some* operations must be inverted (here $+$ to $-$), while other operations (here f) are not affected by the outer data flow direction change and can stay the same. Hence, for this example, inversion is possible since the dependency structure allows us to recover one argument of the addition, and from this we can deduce the other, yielding the inverse function:

$$x_0 = y_0; \quad x_1 = y_1 - f(x_0)$$

3 Preliminaries

In this section, we give a brief introduction to Lean's syntax and introduce the language our algorithm operates over.

3.1 Lean Syntax Overview

Lean is a dependently-typed functional programming language, that can both run programs (via compilation to C), and serve as a proof assistant. In Lean, inductive families are defined with `inductive`, and functions with `def` and **`theorem`**, depending on whether they produce values or proofs. `abbrev` introduce abbreviations. Inductive constructors and pattern matching branches are separated by `|`. Formal arguments are given in parenthesis `(x:a)`, except implicit arguments are given in braces `{x:a}`. Single-letter free variables in the type of a top-level definition are considered implicitly bound by that definition. Dependent product types use the symbol `Σ` when combining two values, `∃` when combining two proofs and `Σ'` when combining a value and a proof, and dependent functions use the symbol `∀`. Given an equality `h: A=B`, a term `a: F A` can be cast into a term `(h▷ a): F B`. In Lean, proof tactics

begin with the keyword by, but for the purposes of this paper, we replace all proofs with ..., as tactics are more useful for interactive exploration rather than reading, even if they are only a few instructions each.

3.2 Signature of Base Language

Our program inversion algorithm is parametrized over a type class Sig that abstracts over the language of the program to be inverted and consists of three parts: syntax and semantics of the language and decidability of equality of syntactical elements:

```
class Sig where
Ty: Type;                 dTy: Ty→Type;                      [eqTy:   DecEq Ty]
Op₀: Ty→Type;             dOp₀: Op₀ α→dTy α;                 [eqOp₀: DecEq (Op₀ α)]
Op₁: Ty→Ty→Type;          dOp₁: Op₁ β α→dTy β→dTy α;         [eqOp₁: DecEq (Op₁ β α)]
Op₂: Ty→Ty→Ty→Type;       dOp₂: Op₂ β γ α→dTy β→dTy γ→dTy α; [eqOp₂: DecEq (Op₂ β γ α)]
```

The syntax of the underlying language consists of some types Ty, nullary operations Op_0, unary operations Op_1 and binary operations Op_2. The semantics of the underlying languages come in the form of functions that map an operation and denotations for their arguments to the denotation of their return type (dTy, dOp_0, dOp_1 and dOp_2). For each syntactical type and operation, we need a way of deciding equality between them (eqTy, eqOp_0, eqOp_1 and eqOp_2).

While Sig specifies the syntax and semantics of the language, our program inversion algorithm also expects a set of sound semi-inverses for each operation of the language, captured by the SemiInv type class (which will be discussed in more detail in Section 5.2). Then, our proofs of sound program inversion work for any set of operations that instantiates both the Sig type class and the SemiInv type class.

The following development is parametrized over an instance s of the Sig type class:

```
variable [s: Sig]
```

3.3 ANF Terms with de Bruijn Indices

We now define ANF terms and evaluation on top of Sig using intrinsically-typed and -scoped syntax [2, 1] with de Bruijn indices because they simplify reasoning about terms, as variable lookup and evaluation become total functions. Hence, a context Ctx is a list of types (with constructors [] and ::). An environment Env Γ for a context Γ is a list of values, one value for each type in the context (with constructors nil and cons):

```
abbrev Ctx := List s.Ty              inductive Env: Ctx → Type
                                       | nil: Env [] | cons: s.dTy α → Env Γ → Env (α::Γ)
```

A variable has type Idx Γ α for a context Γ and a type α, representing an index into the list Γ that can only be constructed if Γ assigns type α to that index, such that the function Idx.eval to look up the value of a variable in an environment is total:

```
inductive Idx: Ctx → s.Ty → Type          def Idx.eval: Idx α Γ → Env Γ → s.dTy α
  | hd: Idx (a::Γ) a                        | hd, cons a _ ⇒ a
  | tl:  Idx Γ a → Idx (b::Γ) a             | tl i, cons _ γ ⇒ i.eval γ
```

A primitive expression Prim Γ α is indexed by a context Γ, in which its variables live, and its type α. Primitives consist of an operation (which can be nullary op_0, unary op_1 or

202 binary op_2) along with the required number of variables that the operation receives as input.
203 A primitive can be evaluated in an environment by looking up the values of the variables
204 and applying the operation denotation via Prim.eval:
205

```
inductive Prim: Ctx → s.Ty → Type
  | op₀: s.Op₁ α → Prim Γ α
  | op₁: s.Op₁ β α → Idx Γ β → Prim Γ α
  | op₂: s.Op₂ β γ α →
        Idx Γ β → Idx Γ γ → Prim Γ α
```

```
def Prim.eval (γ: Env Γ): Prim Γ α → s.dTy α
  | op₀ o ⇒ s.dOp₀ o
  | op₁ o x₁ ⇒ s.dOp₁ o (x₁.eval γ)
  | op₂ o x₁ x₂ ⇒
      s.dOp₂ o (x₁.eval γ) (x₂.eval γ)
```

206 An ANF term Anf Γ α is indexed by a context Γ for its free variables, and a type α. It is
207 either (i) a final return ret x for a variable x or (ii) a binding bnd $\{\alpha_0\}$ p k, consisting of a
208 primitive p of type α_0 to be bound to a variable, and a continuation term k in which the
209 primitive is bound, i.e., the context of the continuation is extended by α_0. Evaluating an
210 ANF term with Anf.eval requires an environment of its free variables and returns a single
211 value:
212

```
inductive Anf: Ctx → s.Ty → Type
  | ret: Idx α Γ → Anf Γ α
  | bnd:
      Prim Γ α₀ → Anf (α₀::Γ) α → Anf Γ α
```

```
def Anf.eval (γ: Env Γ): (t: Anf Γ α) → s.dTy α
  | ret x ⇒ x.eval γ
  | bnd p k ⇒ k.eval (cons (p.eval γ) γ)
```

213 4 From Inverse Functions to Inverse Programs

214 In this section, we define a proof object AnfIsInv to capture the concept of an inverse.

215 The main novel concept in our mechanization is that of a *concordance*, a map from the
216 variables of one program to the variables of another program that only relates variables which
217 evaluate to the same value. To make this notion precise, we first define the notion of a trace,
218 which records the values of variables during program evaluation (Section 4.1). We then define
219 maps from one context to another – renamings (Section 4.2) and weakening (Section 4.3).
220 Consequently, we define concordances, as an extension of a renaming that asserts that
221 variables are mapped to variables that have the same value in any trace (Section 4.4). Finally,
222 we conclude that, if we have an AnfIsInv structure (consisting of a term g and a concordance
223 from g to f that maps g's output to f's input), then g is an inverse of f (Section 4.5). In
224 summary, whenever we have an AnfIsInv structure for a term, we have an ANF term which is
225 the inverse of said term (Theorem 6).

226 4.1 The Relationship between Evaluation and Tracing

227 For constructing a concordance between two terms, it is insufficient to evaluate an ANF
228 term and only look at the final result. We also need to capture and relate *the value of all*
229 *variables* encountered during evaluation. For this we need the defining context Anf.defs, the
230 return variable Anf.retv and the trace Anf.trace of an ANF term, and a lemma about the
231 relationship between them.

232 The defining context Anf.defs is its free variable context concatenated with one type for
233 each binding in the term, and the return variable Anf.retv is simply the variable in the final
234 return clause:

235

```
def Anf.defs: Anf Γ α → Ctx
  | ret _ ⇒ Γ
  | bnd (α₀:=α₀) _ k ⇒ k.defs (α₀::Γ)

def Anf.retv: (t: Anf Γ α) → Idx α t.defs
  | ret x ⇒ x
  | bnd _ k ⇒ k.retv
```

Producing the trace of a term is similar to evaluating it, except we do not just return the final value but instead return the environment (which contains one value for each defined variable):

```
def Anf.trace (γ: Env Γ) (t: Anf Γ α): Env t.defs
  | ret _ ⇒ γ | bnd p k ⇒ k.trace (cons (p.eval γ) γ)
```

For illustration, consider the defining context, return variable, evaluation, and trace of the following example:

```
def sample: Anf [I, I] B := bnd (op₂ add Idx.hd Idx.hd.tl) (bnd (op₁ isZ Idx.hd) (ret Idx.hd))
#eval sample.defs  -- [B, I, I, I]
#eval sample.retv  -- Idx.hd.tl.tl.tl
#eval sample.eval (cons 12 (cons 42 nil))  -- false
#eval sample.trace (cons 12 (cons 42 nil))  -- [false, 54, 12, 42]
```

The term sample has type Anf [I,I] B, meaning that it has two free integer variables, and returns a boolean. It consists of two bindings, binding an integer and a boolean. Its defining context is [B,I,I,I] which consists of the two free variables prepended by one type for each binding in reverse order. The trace contains one value for each variable of the defining context, in order of the defining context. For example, the result of the addition is 54. The return variable is hd.tl.tl.tl, which is the de Bruijn encoding for the fourth variable of the term. We can see that the evaluation of the term is equal to the value of the fourth element (in reverse position) of the trace.

We can generalize this relationship between evaluation and the value of the return variable in the trace of a term with the following lemma, which is proven by a simple induction over the term structure:

▶ **Lemma 1** (Eval is Return of Trace). *The value of a term is equal to value of the return variable in the trace of the term:*

```
theorem Anf.retv_eval_trace (e: Anf Γ α):  e.eval γ = e.retv.eval (e.trace γ)
```

4.2 Renaming and Inverse Renaming

A concordance needs to relate the new term's variables to the old term's variables. The first step towards that is to define a mere renaming – a map from variables in one context to variables in another context (without the proof that they hold the same value).

A renaming allows us to keep track of which new variables correspond to which old variables, and allows us to translate an expression from the old context to the new context, as long as all variables in the expression are part of the renaming. A renaming from a source context Γ to a target context Δ is represented as a value of type Ren Γ Δ and defined as a list of length Γ, containing variables of Δ. The base case, Ren.nil, is the unique renaming from the empty context. The inductive case, Ren.cons, extends a renaming by mapping the head of the source context to a variable in the target context. When a renaming is applied to an index Idx.ren, hd is mapped to the first entry of the renaming, and for every tl an entry in the renaming list is skipped:

275

```
inductive Ren: (Γ Δ: List s.Ty) → Type
  | nil: Ren [] Δ
  | cons: Idx Δ α → Ren Γ Δ → Ren (α::Γ) Δ
```

```
def Idx.ren: Idx Γ α → Ren Γ Δ → Idx Δ α
  | hd, cons r _ ⇒ r
  | tl i, cons _ rs ⇒ i.ren rs
```

276 For example, a renaming of a variable Idx.hd.tl.tl.tl would look up the third entry in the
277 renaming. For illustration, let us briefly compare an example using named and de Bruijn
278 indices. Using names, a possible renaming from a context x : Int, y : Bool, z : String to a
279 context a : Int, b : String, c : Int, d : Bool, e : Bool, would be the mapping $x \mapsto a, y \mapsto d, z \mapsto b$.
280 Using de Bruijn, the renaming is simply a list with one entry for each of the types in the
281 source contexts of positions in the target context:

```
282 example {Int Bool String}: Ren [Int,Bool,String] [Int,String,Int,Bool,Bool] :=
283   cons Idx.hd (cons Idx.hd.tl.tl.tl (cons Idx.hd.tl nil))
```

284 Renamings act on primitives (Prim.ren) by renaming all the contained variables. When
285 applying a renaming to an environment, we look up the renaming's entries (the variables in
286 the target context Δ) in the environment.
287

```
def Prim.ren (r: Ren Γ Δ): Prim Γ α → Prim Δ α
  | op₀ o ⇒ op₀ o | op₁ o x₁ ⇒ op₁ o (x₁.ren r)
  | op₂ o x₁ x₂ ⇒ op₂ o (x₁.ren r) (x₂.ren r)
```

```
def Env.ren (γ: Env Γ): Ren Δ Γ → Env Δ
  | nil ⇒ nil
  | cons x r ⇒ cons (x.eval γ) (γ.ren r)
```

288 To map variables in the target context back to the source context, we define a partial
289 function Ren.iRen that performs the inverse renaming. The function takes a renaming and
290 searches for a variable that is equal to its argument i=i', and returns its position, or none if
291 it cannot be found:

```
292 def Idx.iRen (i: Idx Δ α): Ren Γ Δ → Option (Idx Γ α)
293   | nil ⇒ none | cons (α:=α') i' rs ⇒
294     if h: ∃ (h: α=α'), h▷ i=i' then return h.1▷ hd else return (← rs.iRen i).tl
```

295 This notion of inverse renaming also extends to primitives (Prim.iRen) in a straightforward
296 way:

```
297 def Prim.iRen (r: Ren Γ Δ): Prim Δ α → Option (Prim Γ α)
298   | op₀ o ⇒ return op₀ o | op₁ o x₁ ⇒ return op₁ o (← r.iRen x₁)
299   | op₂ o x₁ x₂ ⇒ return op₂ o (← r.iRen x₁) (← r.iRen x₂)
```

300 Going forward, we need to prove that the inverse renaming is really the inverse of renaming
301 when defined (Idx.ren_iRen, Prim.ren_iRen), and that renaming and evaluation commute
302 with each other (Idx.eval_ren, Prim.eval_ren):

303 ▶ **Lemma 2** (Inverse Renaming Is Inverse Of Renaming)**.**

```
304 theorem Idx.ren_iRen {r: Ren Γ Δ} {x: Idx Γ α} {y: Idx Δ α}:
305   y.iRen r = some x → x.ren r = y
306 theorem Prim.ren_iRen {r: Ren Γ Δ} {p: Prim Γ α} {q: Prim Δ α}:
307   q.iRen r = some p → p.ren r = q
```

308 ▶ **Lemma 3** (Evaluation Commutes With Renaming)**.**

```
309 theorem Idx.eval_ren {r: Ren Δ Γ} {γ: Env Γ} {x: Idx Δ α}:
310   x.eval (γ.ren r) = (x.ren r).eval γ
311 theorem Prim.eval_ren {r: Ren Δ Γ} {γ: Env Γ} {p: Prim Δ α}:
312   p.eval (γ.ren r) = (p.ren r).eval γ
```

313 The proofs for Idx.ren_iRen and Idx.eval_ren proceed by induction on the structure of
314 the renaming. The proof Prim.ren_iRen and Prim.eval_ren follows by structural induction
315 on the primitive, using the previously established theorems for variables.

316 4.3 Weakening Laws

317 All maps between contexts are renamings, but there is one map between contexts that is
318 particularly useful – weakening. Weakening is the well-known type system rule that represents
319 the ability to extend environments of terms while preserving their meaning. For example, a
320 term which is valid in context Γ is also valid in the context (x:α),Γ.

321 Weakening a de Bruijn index into a context with a single additional binding is simply
322 Idx.tl. We also define repeated weakening Idx.wks, which weakens an index i once for each
323 binding in a given term k, such that the index now points into the defining context of said
324 term:

```
325 def Idx.wks (i: Idx Γ α): {k: Anf Γ β} → Idx k.defs α | Anf.ret _ ⇒ i | Anf.bnd _ k ⇒
326     i.tl.wks
```

327 We prove that evaluation commutes with simple and repeated weakening for primitives:

328 ▶ **Lemma 4** (Weakening Laws for Primitives). *Evaluating a primitive in an extended environ-*
329 *ment is the same as evaluating the original in the original environment:*

```
330 theorem Idx.wks_eval_trace {k: Anf Γ β} {x: Idx Γ α}: x.wks.eval (k.trace γ) = x.eval γ
```

331 4.4 Concordances

332 Intuitively, a concordance from f to g is a special renaming that only maps variables which
333 have the same value in any trace.

334 Formally, we first define an equation Equation Γ in context Γ, consisting of a variable in
335 that context as left hand side (lhs), and a primitive expression of the same type ({α}) as
336 right hand side (rhs). Furthermore, we say that an equation eq is *satisfied* by a term e iff
337 the variable and the primitive of the equation evaluate to the same value in any trace of the
338 term (Sat e eq):
339

```
structure Equation (Γ: Ctx): Type where
  {α: s.Ty};  lhs: Idx Γ α;  rhs: Prim Γ α
```

```
def Sat (e: Anf Γ χ): Equation e.defs → Prop
  | ⟨x, p⟩ ⇒ ∀ γ,
      x.eval (e.trace γ) = p.eval (e.trace γ)
```

340 A concordance Conc e e' r extends a renaming r from the free variables of e into the
341 defined variables of e' with one additional entry for each defined variable of e to a defined
342 variable in e'. In particular, for each binding of a primitive p (in e), the renaming is extended
343 using Conc.bnd by a mapping from p to a variable x' (of e'), while asserting that x' and p,
344 when evaluated within the trace of e', have the same value (Sat).

```
345 inductive Conc: (e: Anf Γ χ) → (e': Anf Γ' χ') → Ren Γ e'.defs → Type where
346   | ret {x: Idx Γ χ}: Conc (ret x) e' r
347   | bnd {x': Idx e'.defs α}: Sat e' ⟨x', p.ren r⟩ → Conc e e' (cons x' r) → Conc (bnd p e) e' r
```

348 The renaming r in a concordance Conc e e' r is just a renaming from the *free variables* of
349 e into the defined variables of e', but as each entry of the Conc list extends this renaming
350 with an additional mapping, we can also extract a full renaming from all of the *defined*
351 *variables* of e to the defined variables of e':

```
def Conc.toRen {r: Ren Γ e'.defs}: Conc e e' r → Ren e.defs e'.defs
  | ret _ ⇒ r | bnd _ rs ⇒ rs.toRen
```

In the following lemma, we relate the renaming that can be extracted from a concordance with the renaming that the concordance extends.

▶ **Theorem 5** (Concordance Commutes with Trace). *If two ANF expressions* e *and* e' *are related by a concordance* c *extending the renaming* r*, then renaming the trace of* e' *by* c *is equivalent to taking the trace of* e *after renaming the trace of* e' *by* r*.*

```
theorem Conc.conc_trace {e: Anf Γ χ} {e': Anf Γ' χ'} {r: Ren Γ e'.defs} (c: Conc e e' r):
  ∀ (γ': Env Γ'), (e'.trace γ').ren c.toRen = e.trace ((e'.trace γ').ren r)
```

Proof. The proof is by induction on the concordance structure: In the case ret, the traces trivially coincide. In the case bnd, the property follows by unfolding the trace definitions, applying the satisfyability property of the concordance, and using the properties of renaming and primitive evaluation. ◀

4.5 Inverse Terms Are Inverse

Now, we are ready to capture the concept of an inverse function. To formalize inverse ANF terms, we define a structure AnfIsInv {orig} γ a, where orig represents the original term, γ the input to the new term, and a the output variable of the new term. The structure has three components: the new term, a concordance from the new term to the original extending the renaming γ, and a proof that renaming the term's return variable yields the specified variable a:

```
structure AnfIsInv {orig: Anf E β} (γ: Ren Γ orig.defs) (a: Idx orig.defs α) where
  term: Anf Γ α
  conc: Conc term orig γ
  isInv: term.retv.ren conc.toRen = a
```

More specifically γ is a subset of the defined variables of orig (represented as a renaming), and a is a variable of orig, whose value is to be computed by term.

We prove that an element of AnfIsInv is in fact an inverse of the original program. More precisely: Whenever we have a term f and an AnfIsInv called g, whose input is the output of the original term f.retv and whose output is the input hd.wks of f, then g is an inverse of f, i.e., applying f to any value i can be undone by applying g:

▶ **Theorem 6** (AnfIsInv Implies Sound Inversion).

```
theorem invert_sound (f: Anf [α] β) (g: AnfIsInv [f.retv] hd.wks): ∀ i, i = g.1.eval [f.eval [i]]
```

Proof. By Lemma 1, evaluation is the same as the return of the trace. By Theorem 5 the trace of g on f.eval [i] is a subset of the trace of f on i. By Lemma 3, we can rename the return variable instead of renaming the trace. By AnfIsInv.isInv, the return value of g renames to hd.wks. By Lemma 4, the weakening in hd.wks cancels out with extended tracing to the hd of [i]. Thus, evaluating the inverted term on the output of the original yields the input, as required. ◀

The use of tracing and concordances provides a robust foundation for certified inversion in ANF. This result establishes that constructing an AnfIsInv proof object is sufficient to prove that g is an inverse of f. It is now only left to show that our inversion method does indeed construct such an object.

5 Program Inversion as (De-)Sequentialization

In this section, we define program inversion, which happens in three steps.

- An ANF term is *desequentialized* into a set of equations of the form x = p, where x is a variable and p is a primitive expression (Section 5.1)
- the set of equations is *enriched* (Section 5.2) by adding each equation's semi-inverses.
- the set of equations is *sequentialized* back into an ANF term, by picking one equation after another, starting from the context where only the original term's output is known (Section 5.3)

Finally, we combine these three steps to create a program inversion procedure (Section 5.4).

5.1 Desequentialization

The first step of the inversion procedure is to transform an ANF term into a list of equations. Each binding of an ANF term lives *in a different context*, as their context is extended by every prior binding. To be able to reorder and rewrite the bindings of an ANF term, we desequentialize it into a set of equations, where all variables and primitives live *in the same context*.

The smallest common context to which all bindings of an ANF term can be weakened, is the context containing all definitions of the ANF term, i.e., the defining context. Weakening generalizes from indices to primitives in a natural way (Prim.wk and Prim.wks):

```
def Prim.wk: Prim Γ α → Prim (β::Γ) α
  | op₀ o ⇒ op₀ o | op₁ o x₁ ⇒ op₁ o x₁.tl
  | op₂ o x₁ x₂ ⇒ op₂ o x₁.tl x₂.tl
```

```
def Prim.wks (p: Prim Γ α):
{k: Anf Γ β} → Prim k.defs α
  | op₀ o ⇒ op₀ o | op₁ o x₁ ⇒ op₁ o x₁.wks
  | op₂ o x₁ x₂ ⇒ op₂ o x₁.wks x₂.wks
```

We prove that evaluation commutes with simple and repeated weakening for primitives:

▶ **Lemma 7** (Weakening Laws for Primitives). *Evaluating a primitive in an extended environment is the same as evaluating the original in the original environment:*

```
theorem Prim.wk_eval_cons (p: Prim Γ α):
  p.wk.eval (cons v γ) = p.eval γ
theorem Prim.wks_eval_trace {k: Anf Γ β} {p: Prim Γ α}:
  p.wks.eval (k.trace γ) = p.eval γ
```

Weakening also holds for satisfaction judgments, i.e., if an equation is satisfied by a term, it remains satisfied by the same term with one additional binding:

```
def Sat.wk {k: Anf (α::Γ) χ}: (Σ' e, Sat k e) → Σ' e, Sat (bnd p' k) e
  | ⟨e, h⟩ ⇒ ⟨e, fun γ ⇒ h (cons (p'.eval γ) γ)⟩
```

Now, we can define desequentialization as a function Anf.deseq that generates weakened equations in their common context, and their soundness proofs from an ANF term:

```
def Anf.deseq : (term: Anf Γ χ) → List (Σ' eq, Sat term eq)
  | ret _ ⇒ [] | bnd p k ⇒ ⟨⟨hd.wks, p.wk.wks⟩, by ...⟩ :: k.deseq.map Sat.wk
```

The omitted proof (by ...) that the weakened equations are satisfied by the original term follows from the established weakening laws for Idx.wks_eval_trace, Prim.wk_eval_trace and Prim.wks_eval_trace.

For illustration, let us revisit the example from Section 4.1. The term had two free variables, and two defined variables. Desequentializing it will produce the following two equations in the common context, i.e., one equation for each defined variable:

```
#eval sample.deseq -- [x₂ == x₁ + x₀,  x₃ == iszero x₂]
```

5.2 Enrichment by Semi-Inverses

To be able to invert an ANF term, we cannot just rely on the equations generated from the original program, but we need to *enrich* the set of equations gained from desequentialization of the program by adding their semi-inverses. Then we can construct the inverse of the whole ANF term from the semi-inverses for the individual equations.

As the semi-inverses are specific to the operations of the developer-defined language, which is to be inverted by our algorithm (Section 3.2), the developer needs to provide the semi-inverses of the operations. To ensure that our ANF inversion procedure is sound, our mechanization also requires the developer to provide a proof that the semi-inverses of the equations are valid, i.e., preserve satisfiability with regard to the original term. This requirement is captured by the following type class:

```
class SemiInv where
  semiInv {e: Anf Γ α}: List (Σ' eq, Sat e eq) → List (Σ' eq, Sat e eq)
```

5.3 Sequentialization

In this subsection, we define sequentialization, which turns a set of equations into a well-scoped ANF term. It does this by repeatedly removing an equation from the set of equations, and appending it as a new line of code to the term under construction. To this end, we first define a *sequencing step* that takes and returns a *sequencing state*, and then implement sequentialization as a terminating loop of steps.

Sequentialization constructs ANF terms by appending bindings first-to-last. But similar to the construction of lists with nil and cons, the inductive definition of ANF terms suggests a last-to-first construction as the natural order of construction (starting with the final ret clause and wrapping the term into further bnd clauses to grow a term). To construct terms first-to-last, each step of our sequentialization procedure will emit an individual 'ANF term with a hole at the end', such that appending new bindings corresponds to hole-filling. An 'ANF term with a hole' can be represented by working in continuation-passing-style, where hole-filling is just function composition. The definition AnfContIsInv ins outs is the type representing an AnfIsInv ins x term with an AnfIsInv outs x hole to be filled:

```
abbrev AnfIsInvCont {e: Anf E χ} (ins: Ren Γ e.defs) (outs: Ren Δ e.defs) :=
  ∀ {α} {x: Idx e.defs α}, AnfIsInv outs x → AnfIsInv ins x
```

Note the variance of the arguments: An AnfIsInv ins x over an original program e, is essentially a function ins → x, i.e., knowing the values of the variables ins in the trace of e, we can determine the value of x in the trace of e. As such an ANF continuation AnfIsInv outs x → AnfIsInv ins x is essentially a function of type (outs → x) → (ins → x) which is a function ins → outs in continuation-passing-style, in other words, the meaning of an ANF continuation boils down to: if we know the values of ins in e, we can know the value of outs in e.

Now, a sequencing state SeqSt {orig} ins ins' is parametrized by an original program orig and a subset ins of the variables in the original program (ins' shows that ins is a subset, by mapping each variable to a variable in the original program) which should be the input of the

program to be generated. Because we want to perform inversion, ins will be the set of output variables of the original program, capturing the fact that inversion turns outputs into inputs. The sequencing state consists of the following parts: (i) The ANF term under construction (represented by the ANF continuation res from ins' to outs'), (ii) a set of equations eqs to be used to build the term, and (iii) a proof that the defined variables of the program are a subset of the original program's variables (out, outs').

```
structure SeqSt {orig: Anf E χ}
  {ins: Ctx} (ins': Ren ins orig.defs) -- subset of originals that should be input to res
where
  {out: Ctx}; {out': Ren out' orig.defs}
      -- subset of originals that are currently defined in res
  res: AnfIsInvCont ins' outs' -- term under construction
  eqs: List (Σ' e, Sat orig e) -- the set of equations to construct the term with
```

For each sequencing step we must identify an equation whose left-hand side variable is *not yet* defined and whose right-hand side uses only variables which are *already* defined in the current context. This is done by the function SeqSt.seqStep, which searches for such an equation, removing it from the set of equations and appending it to the program:

```
1 def SeqSt.seqStep {orig: Anf E χ} {ren: Ren Γ orig.defs} (st: SeqSt ren):
2 Option (SeqSt ren) := do
3   let needle := Σ α, (x: Idx orig.defs α) × AnfIsInvCont (cons x st.ren) st.ren
4   let ⟨i, α, x, res'⟩ ← st.eqs.any? (β := needle) fun | ⟨⟨x, p⟩, h⟩ ⇒
5     match x.iRen st.ren with | some _ ⇒ none | none ⇒ -- x is still undefined
6       match h_p: p.iRen st.ren with | none ⇒ none | some p' ⇒ -- p is well-scoped
7         some ⟨_, x, fun ⟨k1, k2, k3⟩ ⇒ ⟨bnd p' k1, bnd (by ...) k2, k3⟩⟩
8   pure {res := st.res ∘ res', eqs := st.eqs.eraseIdx i}
```

The function first searches for any equation ⟨x, p⟩ (Line 3) such that x is not yet defined in the current context, and the variables in p are already defined in the context. These two properties are checked by applying the inverse renaming iRen st.ren: only if the variable x has not been defined the inverse renaming will yield none (Line 4), and only if all variables in the primitive p have already been defined, inverse renaming will yield some p' (Line 5). If such an equation is found, it constructs a new state (Line 7): The continuation res is composed with the new binding. The used equation eqs[i] is removed from the list.

Any equation picked by any? that fulfills these conditions is as good as any other, because by the satisfaction of semi-inversion generated by enrichment for individual operations, all equations that define the same variable will also define the same value. It is also not possible to get stuck by adding an equation for 'the wrong variable', as adding an equation for any variable to the program only ever makes more variables available when appending equations.

Sequencing SeqSt.seq proceeds by repeatedly applying seqStep until no further progress can be made. This function ensures termination by observing that each successful step strictly decreases the number of remaining equations:

```
def SeqSt.seq {orig: Anf E χ} {ren: Ren Γ orig.defs} (st: SeqSt ren): SeqSt ren :=
  match st.seqStep with
  | none ⇒ st
  | some st' ⇒ if st'.eqs.length < st.eqs.length then st'.seq else st'
termination_by st.eqs.length
```

Together, the sequentialization implements a topological sort of the equations, sequencing them in an order that respects dependencies – i.e., a variable can only be bound once all of its dependencies are available in the current context. The explicit tracking of renamings and use of dependent types ensures that only well-scoped and well-typed equations are sequenced.

5.4 Program Inversion

The inversion Anf.semiInvert function takes as an argument the old term to be inverted term: Anf Is O, together with a subset of the free variables of the old term (known: Ren Os term.defs) that should become the input of the new term, and a variable of the old term (i: Idx Is I) that should become the output of the new term.

```
def Anf.semiInvert {Is} {Os} [SemiInv] (e: Anf Is O) (known: Ren Os e.defs) (i: Idx Is I):
Option (AnfIsInv known i.wks) := do
  let st: SeqSt known := SeqSt.seq {
    eqs := SemiInv.semiInv e.deseq
    res := fun k ⇒ k }
  match hi: i.wks.iRen st.ren with | none ⇒ none | some i' ⇒
    return st.res ⟨ret i', ret, Idx.ren_iRen hi⟩
```

This function first initializes the sequencing state with an ‘empty’ ANF term under construction (identity function) res (Line 5), and a set of equations from desequentialization and enrichment of the given term e (Line 4). The sequencing state is sequentialized with SeqSt.seq (Line 3). We check via the generated renaming (st.ren) whether the input variable (i.wks) has been successfully recovered (Line 6), and if so, we complete the inverted term with a final return of the i' variable and correspondingly the concordance with a final ret, while proving the fact that the final value is an inverse to the original term via (Line 7).

We can finally mechanize our earlier, informal statement of program inversion:

$$\forall\, f, g.\ \mathsf{invert}\ f = \mathsf{some}\ g \implies \forall\, x.\, g\,(f\,x) = x$$

To define the invert function used in that statement, known will be initialized with the output of the old term, and i with the input of the old term (note that the Anf.semiInvert function can also construct other semi-inverses by choosing a different variable for i):

```
def Anf.invert [SemiInv] (term: Anf [I] O): Option (Anf [O] I) :=
  return (← term.semiInvert (known:=.cons term.retv .nil) (i:=.hd)).1
```

▶ **Corollary 8** (Correctness of Inversion). *When invert f returns some g, then g inverts f.*

```
theorem Anf.invert_fun_sound [SemiInv] {f: Anf [I] O} {g : Anf [O] I}:
  f.invert = some g → ∀ i, i = g.eval [f.eval [i]]
```

Proof. Follows from Theorem 6. ◀

6 Inverting Normalizing Flows

We demonstrate that our inversion procedure can invert nontrivial and interesting programs by defining several normalizing flow layers as laid out by Prince [19]. They are shown in pseudo code in Figure 1, their implementation can be found in the supplement. Note that the layers take both inputs (labeled $\overrightarrow{x}$), and weights (a tuple of a matrix M and a vector $\overrightarrow{b}$,

Table 1 Exact density inference tools vs. our algorithm for Normalizing Flows.

Layer	SPPL	λPSI	Hakaru	Stochaskell	Ours
Additive Coupling Flow	○	◐	◐	○	●
Residual Flow	○	○	○	○	●
Autoregressive Flow	○	○	◐	◐	●

labeled w). For inverting neural networks, the weights are considered fixed parameters for both the original function and the inverse. So, if the original layer has type $A \times W \to B$, where W is the weights tuple type, the inverse has type $B \times W \to A$. We write $\overrightarrow{x_1 \ldots x_n}$ as the concatenation of vectors $\overrightarrow{x_1}$ to $\overrightarrow{x_n}$

$$\mathsf{linear}\ (M, \vec{b})\ \vec{x} := M \times \vec{x} + \vec{b}$$

$$\mathsf{twoLinear}\ w_1\ w_2\ \vec{x} := \mathsf{linear}\ w_1\ (\mathsf{relu}\ (\mathsf{linear}\ w_2\ \vec{x})) \qquad \mathsf{swap}\ (x_1, x_2) := (x_2, x_1)$$

$$\begin{aligned}
\mathsf{addCoup}\ w_1\ w_2\ (\vec{x_1}, \vec{x_2}) &:= (\vec{x_1} + \mathsf{twoLinear}\ w_1\ w_2\ \vec{x_2},\ \vec{x_2}) \\
\mathsf{residual}\ w_1\ w_2\ w_3\ w_4\ x &:= \mathsf{addCoup}\ w_1\ w_2\ (\mathsf{swap}\ (\mathsf{addCoup}\ w_3\ w_4\ x)) \\
\mathsf{autoreg}\ M_1\ M_2\ M_3\ \vec{b}\ (\vec{x_1}, \vec{x_2}, \vec{x_3}, \vec{x_4}) &:= (\vec{b} + \vec{x_1},\ M_1 \times \vec{x_1} + \vec{x_2}, \\
&\quad M_2 \times \overrightarrow{x_1 x_2} + \vec{x_3},\ M_3 \times \overrightarrow{x_1 x_2 x_3} + \vec{x_4})
\end{aligned}$$

Figure 1 Some helper functions (top), and normalizing flow layers (bottom) which our algorithm produces sound inverses for.

In particular, we have an example for (i) an additive coupling flow, (ii) a residual flow, and (iii) an autoregressive flow. All layers were implemented in our library and our inversion procedure returns inverses for them. There are non-invertible operations (relu and multiplication) in the presented flow layers.

Comparison As normalizing flows are a form of probabilistic modeling [18], an obvious idea is to implement them in an exact probabilistic programming language like λPSI [7], Hakaru [3], Stochaskell [20], or SPPL [21].

However, these existing approaches fail to compute probability densities for the flows we discussed above, as can be seen in Table 1, where we compare the languages to our library. We present the comparison results using three indicators: ● for success, ○ for failure, and ◐ if successful inference requires known weights, necessitating recomputation after each weight update. In the case of Hakaru, we had to change the order of the elements returned by the additive coupling flow to get a result.

Note that our approach focuses on program inversion, while the languages to which we compare predominantly perform probability density calculation. However, given an inverted flow, all that is needed to calculate a probability density is the flow's derivative, which can be easily computed with standard automatic differentiation frameworks. In that sense, program inversion is the "hard part" of density calculation. In fact, the above-mentioned languages already make use of (simpler) forms of program inversion, hence our results suggest that integrating our algorithm into such languages could improve their expressivity, by enabling them to infer densities for more programs.

7 Related Work

A number of reversible programming languages, like Janus [29] and Π [4], follow the *local invertibility* approach: Each basic operation is invertible, so that program inversion is a simple compositional syntax transformation. While this is conceptually elegant, it limits the operations that can be supported, severely hindering the expressiveness of the language. In contrast, we allow operations that are only semi-invertible or even non-invertible. See Thuné et al. [28] for a more in-depth discussion of the relationship between semi-invertible and locally invertible programming.

Mogensen [11] proposed an inversion algorithm for a system of guarded equations (which can be seen as function definitions in, e.g., Haskell). The algorithm was later extended to higher-order functions [12] and array programming [13]. This line of work offers no proof of correctness, let alone mechanization. Our mechanization deals with additional challenges, such as incremental term construction (addressed by continuation-passing-style) and the need to relate two programs in different contexts (addressed by renamings and concordances). An approach inspired by Mogensens semi-inversion approach has been formalized [16], but this paper focussed on supporting recursion, and said paper seems to omit the possibility of considering multiple semi-inverses for a single operation, therefore moving away from the domain that we consider in this paper.

Jeopardy [9] is an invertible programming language that supports locally non-invertible operations. Invertibility is checked using static analysis but compilation or mechanization is not discussed. Sparcl [10] is a higher-order language for semi-invertible programming. It captures invertibility through linear function types and a special invertibility modality. Kalpis [28] is similar to Sparcl, but dispenses with the more complex type system of Sparcl, instead featuring a normal $\to$ and invertible $\leftrightarrow$ function type. Kalpis can also be compiled to a locally invertible combinator language. Compared to our approach, Sparcl and Kalpis can ensure the existence of an inverse through its type system. On the flip side, each semi-inverse of a given operation corresponds to a different type: This means that, when using an operation, the developer already has to decide which semi-inverse they want to use for inverting the program. In our approach, the developer writes unannotated programs and the relevant semi-inverse is chosen by the algorithm.

The inversion of newly-defined normalizing flow layers is typically done manually (See Section 6). In contrast, exact probabilistic programming languages like Hakaru [14, 22, 15], SPPL [21], and λPSI [6, 7] try to *automatically* perform probability density calculation. For inferring the density function, they also need to perform some form of program inversion. But this is tightly coupled to other aspects of probability inference and there is no formal definition and soundness proof of the employed inversion. Moreover, their inversion algorithms are more restricted than our semi-inversion algorithm (e.g., SPPL only inverts sequences of invertible functions), and they have not been applied to normalizing flows.

8 Conclusion

This paper introduced a novel, mechanized approach to semi-inversion for simply-typed total functional programming languages, providing the first formalized and mechanized proof of correctness for semi-inversion. The formalization in Lean ensures that every generated semi-inverse is sound with respect to the original program semantics (assuming the requirements outlined in Section 3 are fulfilled). The mechanized proof increases confidence in the reliability of the approach, and we have also demonstrated that our verified procedure produces inverses for nontrivial programs, namely a number of normalizing flows.

629 ## Declaration on GenAI tools

630 We have used GenAI for turning bullet points into text, and for creating natural language
631 descriptions from code, and for shortening and rephrasing text occassionally, or suggesting
632 alternative formulations, and for latex help. But generally speaking, a human has to rewrite
633 the text afterwards, multiple times, as AI tends to generate very verbous, shallow and often
634 plainly wrong text.

635 ## References

636 **1** Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser.
637 Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program.*
638 *Lang.*, 2(POPL), December 2017. doi:10.1145/3158104.

639 **2** Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term
640 representations in coq. *J. Autom. Reason.*, 49(2):141–159, 2012. URL: https://doi.org/10.
641 1007/s10817-011-9219-0, doi:10.1007/S10817-011-9219-0.

642 **3** Sooraj Bhat, Johannes Borgström, Andrew D. Gordon, and Claudio V. Russo. Deriving
643 probability density functions from probabilistic functional programs. *Log. Methods Comput.*
644 *Sci.*, 13(2), 2017. doi:10.23638/LMCS-13(2:16)2017.

645 **4** Vikraman Choudhury, Jacek Karwowski, and Amr Sabry. Symmetries in reversible pro-
646 gramming: from symmetric rig groupoids to reversible programming languages. *Proc. ACM*
647 *Program. Lang.*, 6(POPL):1–32, 2022. doi:10.1145/3498667.

648 **5** Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling
649 with continuations. In Robert Cartwright, editor, *Proceedings of the ACM SIGPLAN'93*
650 *Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New*
651 *Mexico, USA, June 23-25, 1993*, pages 237–247. ACM, 1993. doi:10.1145/155090.155113.

652 **6** Timon Gehr, Sasa Misailovic, and Martin T. Vechev. PSI: exact symbolic inference for
653 probabilistic programs. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided*
654 *Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23,*
655 *2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 62–83.
656 Springer, 2016. doi:10.1007/978-3-319-41528-4_4.

657 **7** Timon Gehr, Samuel Steffen, and Martin T. Vechev. λpsi: exact inference for higher-order
658 probabilistic programs. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of*
659 *the 41st ACM SIGPLAN International Conference on Programming Language Design and*
660 *Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 883–897. ACM, 2020.
661 doi:10.1145/3385412.3386006.

662 **8** Roshan P. James and Amr Sabry. Information effects. In John Field and Michael Hicks, editors,
663 *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming*
664 *Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 73–84.
665 ACM, 2012. doi:10.1145/2103656.2103667.

666 **9** Joachim Tilsted Kristensen, Robin Kaarsgaard, and Michael Kirkedal Thomsen. Jeopardy:
667 An invertible functional programming language. In Torben Ægidius Mogensen and Lukasz
668 Mikulski, editors, *Reversible Computation - 16th International Conference, RC 2024, Toruń,*
669 *Poland, July 4-5, 2024, Proceedings*, volume 14680 of *Lecture Notes in Computer Science*,
670 pages 124–141. Springer, 2024. doi:10.1007/978-3-031-62076-8_9.

671 **10** Kazutaka Matsuda and Meng Wang. Sparcl: A language for partially invertible computation.
672 *J. Funct. Program.*, 34, 2024. URL: https://doi.org/10.1017/s0956796823000126, doi:
673 10.1017/S0956796823000126.

674 **11** Torben Æ. Mogensen. Semi-inversion of guarded equations. In Robert Glück and Michael R.
675 Lowry, editors, *Generative Programming and Component Engineering, 4th International*
676 *Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005, Proceedings*,

677 volume 3676 of *Lecture Notes in Computer Science*, pages 189–204. Springer, 2005. doi:
678 10.1007/11561347_14.
679 **12** Torben Æ. Mogensen. Semi-inversion of functional parameters. In Robert Glück and Oege
680 de Moor, editors, *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation*
681 *and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA,*
682 *January 7-8, 2008*, pages 21–29. ACM, 2008. doi:10.1145/1328408.1328413.
683 **13** Torben Ægidius Mogensen. Reversible functional array programming. In Shigeru Yamashita
684 and Tetsuo Yokoyama, editors, *Reversible Computation - 13th International Conference, RC*
685 *2021, Virtual Event, July 7-8, 2021, Proceedings*, volume 12805 of *Lecture Notes in Computer*
686 *Science*, pages 45–63. Springer, 2021. doi:10.1007/978-3-030-79837-6_3.
687 **14** Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov.
688 Probabilistic inference by program transformation in hakaru (system description). In Oleg
689 Kiselyov and Andy King, editors, *Functional and Logic Programming - 13th International*
690 *Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, volume 9613 of *Lecture*
691 *Notes in Computer Science*, pages 62–79. Springer, 2016. doi:10.1007/978-3-319-29604-3\
692 _5.
693 **15** Praveen Narayanan and Chung-chieh Shan. Symbolic conditioning of arrays in probabilistic
694 programs. *Proc. ACM Program. Lang.*, 1(ICFP):11:1–11:25, 2017. doi:10.1145/3110255.
695 **16** Naoki Nishida and Germán Vidal. Program inversion for tail recursive functions. In Manfred
696 Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting*
697 *Techniques and Applications, RTA 2011, Novi Sad, Serbia, May 30 - June 1, 2011*, volume 10
698 of *LIPIcs*, pages 283–298. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011. URL:
699 https://doi.org/10.4230/LIPIcs.RTA.2011.283, doi:10.4230/LIPICS.RTA.2011.283.
700 **17** George Papamakarios, Eric T. Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and
701 Balaji Lakshminarayanan. Normalizing flows for probabilistic modeling and inference. *J.*
702 *Mach. Learn. Res.*, 22:57:1–57:64, 2021. URL: http://jmlr.org/papers/v22/19-1028.html.
703 **18** George Papamakarios, Eric T. Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and
704 Balaji Lakshminarayanan. Normalizing flows for probabilistic modeling and inference. *J. Mach.*
705 *Learn. Res.*, 22:57:1–57:64, 2021. URL: https://jmlr.org/papers/v22/19-1028.html.
706 **19** Simon JD Prince. *Understanding deep learning.* MIT press, 2023.
707 **20** David A. Roberts, Marcus Gallagher, and Thomas Taimre. Reversible jump probabilistic
708 programming. In Kamalika Chaudhuri and Masashi Sugiyama, editors, *The 22nd International*
709 *Conference on Artificial Intelligence and Statistics, AISTATS 2019, 16-18 April 2019, Naha,*
710 *Okinawa, Japan*, volume 89 of *Proceedings of Machine Learning Research*, pages 634–643.
711 PMLR, 2019. URL: http://proceedings.mlr.press/v89/roberts19a.html.
712 **21** Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. SPPL: probabilistic programming
713 with fast exact symbolic inference. In Stephen N. Freund and Eran Yahav, editors, *PLDI*
714 *'21: 42nd ACM SIGPLAN International Conference on Programming Language Design and*
715 *Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 804–819. ACM, 2021.
716 doi:10.1145/3453483.3454078.
717 **22** Chung-chieh Shan and Norman Ramsey. Exact bayesian inference by symbolic disintegration.
718 In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN*
719 *Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January*
720 *18-20, 2017*, pages 130–144. ACM, 2017. doi:10.1145/3009837.3009852.
721 **23** The Deepmind Team. Distrax. https://github.com/google-deepmind/distrax. Accessed:
722 2026-20-02.
723 **24** The Oryx Authors. Oryx. https://github.com/jax-ml/oryx, 2022. Accessed: 2025-07-09.
724 **25** The TensorFlow Team. Pyro distributions transforms. https://docs.pyro.ai/en/dev/
725 distributions.html#transforms. Accessed: 2026-20-02.
726 **26** The TensorFlow Team. Tensorflow probability. https://github.com/tensorflow/
727 probability. Accessed: 2026-20-02.

728 **27** The TuringLang Team. Bijectors.jl. `https://github.com/TuringLang/Bijectors.jl`. Ac-
729 cessed: 2026-20-02.
730 **28** Anders Ågren Thuné, Kazutaka Matsuda, and Meng Wang. Reconciling partial and local
731 invertibility. In Stephanie Weirich, editor, *Programming Languages and Systems - 33rd*
732 *European Symposium on Programming, ESOP 2024, Held as Part of the European Joint*
733 *Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg,*
734 *April 6-11, 2024, Proceedings, Part II*, volume 14577 of *Lecture Notes in Computer Science*,
735 pages 59–89. Springer, 2024. `doi:10.1007/978-3-031-57267-8_3`.
736 **29** Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Principles of a reversible pro-
737 gramming language. In Alex Ramírez, Gianfranco Bilardi, and Michael Gschwind, editors,
738 *Proceedings of the 5th Conference on Computing Frontiers, 2008, Ischia, Italy, May 5-7, 2008*,
739 pages 43–54. ACM, 2008. `doi:10.1145/1366230.1366239`.