A Mechanized Formalization of MicroKanren in Agda

Eduardo Henke
Programa de Pós-Graduação em Ciência da Computação,
Universidade Federal de Ouro Preto
Ouro Preto, Brazil
eduardo.henke@aluno.ufop.edu.br

Rodrigo Ribeiro
Programa de Pós-Graduação em Ciência da Computação,
Universidade Federal de Ouro Preto
Ouro Preto, Brazil
rodrigo.ribeiro@ufop.edu.br

Abstract

We present a machine-checked formalization of microKanren in the Agda proof assistant. Building on McBride's accumulating most-general unifier, we develop a coinductive stream model for nondeterministic search with machine-verified soundness, completeness, and maximality for the unification goal. The stream type uses an inverted mutual `record`/`data` pair: a coinductive record `KStream` whose sole observation unfolds to an inductive `Step` with constructors `fail`, `yield`, and `later`. Corecursive definitions are written as copattern equations on `.step`; Agda's guardedness checker accepts both fair interleaving and fair binding entirely without pragmas. The formalization is validated on two case studies: a logic puzzle and an infinite enumeration program. The entire development type-checks in the safe fragment of Agda 2.7.0.1 with the standard library 2.2.

Keywords

Agda, MicroKanren, Dependent Types, Coinduction, Unification

1 Introduction

MiniKanren is a family of relational programming languages pioneered by Byrd and Friedman [6]. Its minimal kernel, *microKanren* [7], distills the essence of relational search into four primitive combinators: a unification goal (≡), a fresh-variable introducer (`callFresh`), disjunction (`disj`), and conjunction (`conj`). Despite microKanren's conceptual simplicity, giving it a formal, machine-checked semantics is non-trivial: the search procedure manipulates substitutions that must remain type-consistent as new variables are introduced, and the resulting stream of answers must be produced *fairly*, i.e., the search must not starve any branch.

This paper describes a full formalization of microKanren in Agda, targeting two goals. First, we want the *implementation* to be type-safe in a strong sense: substitutions carry their scope at the type level, so that applying substitution to a term of the wrong arity is a compile-time error, not a runtime failure. Second, we want the central correctness property of the unification primitive to be formally stated and proved: the substitution returned by `amgu` is a sound and most-general unifier whenever one exists. More specifically, we made the following contributions:

- We developed a fully type-safe, machine-checked formalization of microKanren in Agda 2.7.0.1, in which substitutions and search states are indexed by scope at the type level, ruling out variable-scope errors at compile time.
- We port McBride's formalization of first-order unification by structural recursion [12] to the current version of Agda and its standard library (Agda 2.7.0.1, standard library 2.2).
- We formalize microKanren's search strategy using Agda's support for coinductive types, apply it to solve a logic puzzle that reproduces the structure of the Racket miniKanren implementation, and demonstrate the coinductive machinery on an infinite domain via a pragma-free natural-number generator.
- We introduce `callFresh2`, a combinator that allocates two fresh variables in the same scope, resolving the scope-index mismatch that arises from nesting `callFresh` calls.

The complete formalization was carried out in Agda 2.7.0.1 with the Agda Standard Library version 2.2. In what follows, we describe the main data structures and a selection of functions. For theorems, we omit the Agda code, as it would distract the reader with syntactic details. Instead, we state the main results and provide textual proof sketches, each accompanied by a link to the companion artifact.

The rest of this work is organized as follows: Section 2 provides background on Agda, the microKanren calculus, and the formalization of first-order unification, including the `AmguResult` correctness framework and three machine-verified theorems (soundness, completeness, and maximality). Section 3 presents the microKanren formalization. Sections 4 and 5 describe two case studies using the formalization. Section 6 discusses key design decisions. Section 7 surveys related work, and Section 8 concludes.

2 Background

2.1 The Agda Programming Language

Agda is a dependently typed functional programming language and proof assistant [13]. It is based on Martin-Löf type theory [11], which means that types are first-class values and may depend on other values. This makes it possible to express precise specifications directly in the type language and to enforce them at compile time.

Data Types. Agda data types are introduced with the `data` keyword using GADT syntax. As a running example, consider the Peano natural numbers and the finite type `Fin`:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

Because the index `n` changes in the `suc` constructor, `Fin` is an *indexed family* rather than a simple type.

Propositions as Types. Martin-Löf type theory identifies propositions with types and proofs with terms: the *Curry–Howard correspondence*. A value of type `t ≡ u` is evidence that `t` and `u` are definitionally equal; the sole constructor `refl` witnesses reflexivity. A theorem is a function that, given the appropriate hypotheses as arguments, produces such evidence. This style (programs and proofs written in the same language, verified by the same type checker) is what makes Agda suitable for formalizing algorithms such as the microKanren search engine described in this paper.

Induction as Recursion. A central consequence of the Curry–Howard correspondence is that *proof by induction is definitionally the same as structural recursion*. For an inductive type such as ℕ, to prove a property `P : ℕ → Set` for every natural number it suffices to provide a base case `P zero` and a step function `P n → P (suc n)`: these are exactly the two clauses of a recursive function. Agda's termination checker enforces that the recursion is structural, meaning each recursive call must be on a strict subterm, which corresponds precisely to the well-foundedness condition required by induction.

As a concrete illustration, consider the statement that right-addition of `zero` is the identity:

```
+-identityʳ : ∀ (n : ℕ) → n + zero ≡ n
+-identityʳ zero    = refl
+-identityʳ (suc n) = cong suc (+-identityʳ n)
```

The clause `+-identityʳ zero = refl` is the base case: Agda reduces `zero + zero` to `zero` by the first equation of `_+_`, so `refl` suffices. The clause `+-identityʳ (suc n)` is the inductive step: the inductive hypothesis is the recursive call `+-identityʳ n`, which has type `n + zero ≡ n`, and `cong suc` lifts it to `suc (n + zero) ≡ suc n`. Agda reduces `suc n + zero` to `suc (n + zero)` definitionally, so no explicit rewrite is needed. The termination checker accepts the definition because `n` is a strict subterm of `suc n`. This pattern, one clause per constructor with recursive calls on subterms, pervades every inductive proof in this paper. Readers unfamiliar with Agda are referred to [14, 17].

Coinduction and Copatterns. Coinductive types model potentially *infinite* data, such as streams or non-terminating computations. Where inductive types are defined by constructors (how values are *built*), coinductive types are defined by *observations* (how values are *consumed*). Agda supports coinduction through `record` types marked `coinductive`: the record's field is an observation, and a value of such a type is accepted if every observation eventually returns a result.

As a concrete example, consider an infinite stream of values:

```
record ∞Stream (A : Set) : Set where
  coinductive
  field force : Stream A

data Stream (A : Set) : Set where
  []    : Stream A
  _::_  : A → ∞Stream A → Stream A
```

The `coinductive` keyword marks `∞Stream` as a type whose inhabitants need not be finite. Its sole field `force` computes the next `Stream` step on demand. The paired `data` type `Stream` is the strict layer: it either halts (`[]`) or yields one element followed by a *suspended* tail of type `∞Stream A`.

A corecursive function is defined by specifying what each observation returns, using a *copattern* on the left-hand side:

```
mutual
  repeat : {A : Set} → A → Stream A
  repeat x = x :: ∞repeat x

  ∞repeat : {A : Set} → A → ∞Stream A
  ∞repeat x .force = repeat x
```

The clause `∞repeat x .force = repeat x` reads: "when the `force` observation is applied to `∞repeat x`, the result is `repeat x`". The copattern `.force` on the left-hand side acts as a guard [1]: Agda's guardedness checker accepts the definition because the corecursive call `repeat x` appears exactly one `.force`-step deep, ensuring that every finite sequence of observations terminates. This is the coinductive dual of the inductive requirement that every recursive call be on a strict subterm: instead of *consuming* a smaller argument, a corecursive function must *produce* one observation before recurring.

This `data`/`record` mutual pattern illustrates the coinduction mechanism Agda provides. The microKanren formalization uses an *inverted* variant (Section 3.2): the coinductive `record KStream` is the outer type, and its sole field `step` forces an inductive `Step`—eliminating the separate suspension wrapper while keeping every corecursive call directly under a `Step` constructor.

2.2 The MicroKanren Calculus

microKanren [7] is a minimal functional core for relational programming. The entire system fits in under forty lines of Scheme and relies on three concepts: *states*, *goals*, and *streams*.

States. A state is a pair (s, c) where s is a substitution and c is a counter tracking the next fresh variable index.

Goals. A goal is a function from states to streams of states: $Goal = State \to Stream\ State$. The four primitive goals are: (i) `(== u v)`, the unification goal; (ii) `(call/fresh f)`, the fresh-variable introducer; (iii) `(disj` g_1 g_2`)`, fair interleaving of two streams; (iv) `(conj` g_1 g_2`)`, sequential composition.

Streams. microKanren uses three kinds of streams: the empty stream `mzero`, a mature stream `(cons state rest)`, and an immature stream (a zero-argument thunk). The thunk form enables fair search: when `mplus` encounters a thunk in its first argument, it suspends it and processes the second argument first.

```
(define (mplus $1 $2)
  (cond ((null? $1) $2)
        ((procedure? $1) (lambda () (mplus $2 ($1))))
        (else (cons (car $1) (mplus (cdr $1) $2)))))
```

This Scheme presentation serves as the semantic reference for the Agda formalization developed in Section 3. The central differences are: (i) terms carry their variable scope as a type-level index; (ii) substitutions are scope-indexed `ALists`; and (iii) the stream type is a mutual `data`/`record` pair modelled on the Delay Monad [5], replacing untyped thunks.

2.3 First-Order Unification By Structural Recursion

To make the termination of the unification algorithm structurally apparent, we represent variables using de Bruijn indices.

First-order terms with De Bruijn variables. We work with a three-constructor first-order term language parameterized over an atomic value type `A` with decidable equality:

```
data Term (n : ℕ) : Set where
  var   : Fin n → Term n
  val   : A     → Term n
  ⟨_,_⟩ : Term n → Term n → Term n
```

Variables are represented as elements of `Fin n`, the finite type of natural numbers strictly less than n. This *well-scoped de Bruijn* representation ensures that a `Term n` has at most n distinct variables, and that scope violations are ruled out by type-checking.

The substitution action is captured by `walk`:

$$\texttt{walk} : \texttt{Term}\, m \to (\texttt{Fin}\, m \to \texttt{Term}\, n) \to \texttt{Term}\, n$$

A substitution from m to n is simply a function $\texttt{Fin}\, m \to \texttt{Term}\, n$. Substitution composition $f \bullet g$ is defined as $(f \triangleleft) \circ g$ (pointwise extension of f to terms) and is provably associative via `walk-assoc`.

Association Lists as Substitutions. Following McBride [12], substitutions are represented as *association lists*:

```
data AList : ℕ → ℕ → Set where
  nil       : AList n n
  _snoc_/_  : AList m n → Term m → Fin (suc m)
              → AList (suc m) n
```

The type `AList m n` denotes a substitution whose domain has m variables and whose codomain has n variables. Each `snoc` step eliminates one variable, which is the key insight that makes termination of the unification algorithm structurally apparent.

The Accumulating MGU Algorithm. The accumulating most-general unifier `amgu` takes two terms and a partial substitution (the *accumulator*) and either extends the accumulator to unify the terms or fails:

$$\texttt{amgu} : (t\ u : \texttt{Term}\, m) \to \exists(\texttt{AList}\, m) \to \texttt{Maybe}(\exists(\texttt{AList}\, m))$$

The algorithm proceeds by simultaneous recursion on the structures of t and u. The key cases are: *flex-flex* (two variables), *flex-rigid* (variable vs. non-variable, with occurrence check), *rigid-rigid* (two atoms), and *pair* (structural decomposition). The `snoc` step is handled by the equation:

$$\begin{aligned}&\texttt{amgu}\ t\ u\ (\rho\ \texttt{snoc}\ r/z) = \\ &\texttt{amgu}\ ((r\ \texttt{for}\ z) \triangleleft t)\ ((r\ \texttt{for}\ z) \triangleleft u)\ \rho \gg\!= (\texttt{snoc}\ r/z)\end{aligned}$$

formalized as `amgu-snoc-eq`, which requires careful case analysis because Agda's pattern-match tree for `amgu` dispatches on the natural number component of the accumulator before the list component.

2.3.1 Correctness of Unification.

The Property Framework. We define a *property* of substitutions as a predicate over all substitutions with a given source arity:

$$\texttt{Property}\, m = \forall\{n\}.\ (\texttt{Fin}\, m \to \texttt{Term}\, n) \to \texttt{Set}$$

Key instances are:

- `Unifies` $t\ u\ f \equiv$ `walk` $t\ f \equiv$ `walk` $u\ f$: f unifies t and u.
- `Nothing` P: no substitution satisfies P.
- `Max` $P\ f$: f satisfies P and is more general than every other solution.

The preorder $f \leq g$ says that f is more specific than g: $\exists h,\ \forall x,\ f\, x \equiv (h \bullet g)\, x$.

The `AmguResult` Type. The correctness of `amgu` is expressed via an indexed data type that serves as a *proof-carrying result*:

```
data AmguResult (t u : Term m) (ρ : AList m l)
    : Maybe (∃ (AList m)) → Set where
  unifies     : ∀ {σ++ρ} (σ : AList l n)
                → σ++ρ ≡ σ ++' ρ
                → Max (Unifies t u [-⋄ sub ρ]) (sub σ)
                → AmguResult t u ρ (just (n , σ++ρ))
  not-unifies : Nothing (Unifies t u [-⋄ sub ρ])
                → AmguResult t u ρ nothing
```

Here $P\,[- \bullet f]$ denotes $\lambda g \to P\,(g \bullet f)$, i.e., P relativized through the accumulator f. The `unifies` constructor witnesses that `amgu` returned $\sigma +\!\!+ \rho$, the concatenation of a new partial unifier σ (whose `sub` is the MGU of t and u relative to ρ) with the original accumulator. The `not-unifies` constructor witnesses that no unifier exists.

The main correctness theorem is:

```
amgu-correct : ∀ {l} (t u : Term m) (ρ : AList m l)
  → AmguResult t u ρ (amgu t u (l , ρ))
```

The proof is by simultaneous induction on t, u, and ρ. In the rigid-rigid case ($t = \texttt{val}\ a$, $u = \texttt{val}\ b$) the call succeeds exactly when $a = b$, giving the identity as the trivial MGU. In the flex-rigid cases (a variable against a non-variable term) the occurs check is applied: if the variable occurs in the other term the call fails; otherwise the singleton substitution $x \mapsto t'$ is the MGU, by the variable-elimination lemma. The pair case $\langle t_1, t_2\rangle$ vs. $\langle u_1, u_2\rangle$ is handled by the Optimist Lemma: compute the MGU σ_1 of t_1 and u_1, then compute the MGU σ_2 of t_2 and u_2 using σ_1 as accumulator; their composition $\sigma_2 +\!\!+ \sigma_1$ is the MGU of the pair. The snoc case $\rho = \rho_0\ \texttt{snoc}\ r/z$ uses the identity $\texttt{amgu}\ t\ u\ (\rho_0\ \texttt{snoc}\ r/z) = \texttt{amgu}\ ((r/z) \triangleleft t)\ ((r/z) \triangleleft u)\ \rho_0 \gg\!= (\texttt{snoc}\ r/z)$: the call reduces to an inner call on the pre-substituted terms with accumulator ρ_0, and appending r/z back to the inner result recovers the original accumulator shape; the correctness certificate for the inner call then carries over directly.

Three correctness theorems follow directly from `amgu-correct`:

Theorem 2.1 (Soundness). *If* $\texttt{amgu}\ t\ u\ (k, \sigma)$ *returns* $\texttt{just}\ (k', \sigma')$, *then* $\texttt{sub}\ \sigma'$ *unifies* t *and* u.

Proof. The `unifies` case gives τ with $\sigma' = \tau +\!\!+ \sigma$ and $\texttt{sub}\,\tau \bullet \texttt{sub}\,\sigma$ unifying t and u. The identity $\texttt{sub}(\tau +\!\!+ \sigma) = \texttt{sub}\,\tau \bullet \texttt{sub}\,\sigma$ then gives the result. □

Theorem 2.2 (Completeness). *Whenever* t *and* u *are unifiable, then* $\texttt{amgu}\ t\ u\ (c, \texttt{nil})$ *does not fail.*

Proof. Were the call to return `nothing`, the `not-unifies` case would give a witness that no substitution unifies t and u relative to $\texttt{sub}\,\texttt{nil} = \text{id}$, i.e., no substitution unifies them at all; this contradicts the hypothesis. □

Theorem 2.3 (Maximality (MGU)). *If* $\texttt{amgu}\ t\ u\ (c, \texttt{nil})$ *returns* $\texttt{just}\ (k', \sigma')$, *then* $\texttt{sub}\ \sigma'$ *is a most-general unifier of* t *and* u.

Proof. The `unifies` case with $\rho = \texttt{nil}$ gives $\sigma' = \delta$ with $\texttt{sub}\,\delta$ being the MGU of t and u relative to $\texttt{sub}\,\texttt{nil} = \mathrm{id}$. Since $g \bullet \mathrm{id} = g$ for every g, this is the ordinary MGU condition: $\texttt{sub}\,\sigma'$ unifies t and u, and every unifier f' satisfies $f' = h \bullet \texttt{sub}\,\sigma'$ for some h. □

3 The MicroKanren Formalization

3.1 Search State

A *state* encodes the current knowledge of the search: a counter c of created variables, a count k of remaining free variables in the accumulator, and a partial substitution σ : `AList` c k:

```
record State : Set where
  constructor mkState
  field {scope} {vars} : ℕ
        σ : AList scope vars
```

The scope counter and variable count are implicit record fields, exposed by pattern-matching as `mkState {c₁} {k} σ`. This allows, for instance, `_≡ₖ_` to recover the proof that the state's scope index matches the type index of its term arguments via a decidable equality check. The initial state is `emptyState = mkState nil`.

3.2 Coinductive Streams

Non-deterministic search produces a *stream* of states, potentially infinite. We represent streams using a mutual `record`/`data` pair in which the roles are *inverted* compared to the Delay Monad pattern of Section 2.1:

```
mutual
  record KStream : Set where
    coinductive
    field step : Step

  data Step : Set where
    fail  : Step
    yield : State → KStream → Step
    later : KStream → Step
```

`KStream` is the coinductive record: its sole field `step` forces one observation and returns a `Step`. `Step` is an ordinary inductive type with three constructors: `fail` is the empty stream; `yield s xs` produces state s and a continuation `KStream xs`; `later xs` is a computation step with no output, enabling suspension.

A corecursive function is defined by a copattern equation on `.step`: writing `f ... .step = e` specifies the observation. The right-hand side `e` is a `Step` value, and any recursive calls to `f` appear directly under a `Step` constructor—one guarded step from the copattern boundary. Agda's checker accepts every corecursive function in the formalization this way, without any pragma.

3.3 Stream Primitives

Fair Interleaving. *Fair interleaving* ensures that no branch of the search is starved:

```
interleave : KStream → KStream → KStream
interleave xs ys .step with xs .step
... | fail         = ys .step
... | yield s xs'  = yield s (interleave ys xs')
... | later xs'    = later (interleave ys xs')
```

The copattern `.step` on the left-hand side defines `interleave` as a corecursive function: each `.step` observation reduces to a `Step` value. The three cases serve distinct roles. *Base case:* when `xs` is `fail`, the second stream `ys` is returned unchanged. *Yield case:* the head state `s` is emitted and the roles of the two streams are *swapped*: `ys` becomes the left argument and the tail `xs'` the right. *Later case:* the suspension is preserved and the streams are again swapped. The swap is the mechanism of fairness: after at most one step from the left stream, control passes to the right, and vice versa.

Agda's guardedness checker accepts this definition without any pragma or mutual block: the two recursive calls `interleave ys xs'` appear directly under `yield` and `later`, which are constructors of `Step`—the right-hand side of a copattern `.step` equation.

Binding. Binding applies a goal to each element of a stream and interleaves the results. The implementation uses a *round-robin queue* to track in-flight sub-streams:

```
bindDriver : KStream → List KStream → Goal → KStream
bindDriver xs []        g .step with xs .step
... | fail = fail
... | later xs′ = later (bindDriver xs′ [] g)
... | yield s xs′ = later (bindDriver xs′ [ g s ] g)
bindDriver xs (q ∷ qs′) g .step with xs .step | q .step
... | fail  | fail = later (bindDriver mzero qs′ g)
... | fail  | yield t q′
            = yield t (bindDriver mzero (qs′ ∷ʳ q′) g)
... | fail  | later q′
            = later (bindDriver mzero (qs′ ∷ʳ q′) g)
... | later xs′ | fail = later (bindDriver xs′ qs′ g)
... | later xs′ | yield t q′
            = yield t (bindDriver xs′ (qs′ ∷ʳ q′) g)
... | later xs′ | later q′
            = later (bindDriver xs′ (qs′ ∷ʳ q′) g)
... | yield s xs′ | fail
            = later (bindDriver xs′ (qs′ ∷ʳ g s) g)
... | yield s xs′ | yield t q′
            = yield t (bindDriver xs′ ((qs′ ∷ʳ q′) ∷ʳ g s) g)
... | yield s xs′ | later q′
            = later (bindDriver xs′ ((qs′ ∷ʳ q′) ∷ʳ g s) g)

bindStep : KStream → Goal → KStream
bindStep xs g = bindDriver xs [] g
```

`bindDriver xs qs g` maintains two sources of pending work: the *main stream* `xs` and a *queue* `qs` of in-flight sub-streams produced by earlier applications of `g` to states from `xs`. At each step, `bindDriver` inspects the next constructor of both `xs` and the queue head simultaneously (when the queue is non-empty), dispatching on all nine combinations.

When the queue is empty: `fail` in `xs` terminates; a `later` defers one step; `yield s xs'` enqueues `g s` and emits a `later` to maintain one-step progress. When the queue has a head `q`: a `yield t q'` from `q` emits `t` immediately and re-enqueues the tail `q'` at the back; a `later q'` or `fail` advances or discards `q`. In every case, `xs` advances by one constructor—guaranteeing that every state in `xs` is eventually processed regardless of how many sub-streams are queued.

Agda's guardedness checker accepts `bindDriver` because every recursive call appears directly as the argument of `later` or `yield` in `Step`—the right-hand side of a copattern `.step` equation. No pragma is needed. `bindStep` starts `bindDriver` with an initially empty queue.

3.4 The Four MicroKanren Primitives

Having established the coinductive stream and its core operations, we now define the four goal-level primitives that constitute microKanren.

Unification Goal. The goal $t \equiv_k u$ invokes `amgu` on the current state:

```
_≡ₖ_ : ∀ {c} → Term c → Term c → Goal
_≡ₖ_ {c} t u (mkState {c₁} {k} σ) with Nat._≟_ c₁ c
... | no  _    = mzero
... | yes refl with amgu t u (k , σ)
...   | nothing        = mzero
...   | just (k' , σ') = unit (mkState σ')
```

The runtime check $c_1 \stackrel{?}{=} c$ recovers the proof $c_1 \equiv c$ needed to make σ : `AList` $c\ k$ compatible with t, u : `Term` c. Under the invariant that goals are only applied to states whose counter matches the term's scope (an invariant maintained by `callFresh`), the `no` branch is unreachable.

Fresh Variable Introduction. The `callFresh` combinator introduces a new logic variable:

```
callFresh : (∀ {c} → Term (suc c) → Goal) → Goal
callFresh f (mkState {c} σ) =
  f (var (fromℕ c)) (mkState (liftAList σ))
```

The fresh variable is `var (fromℕ c)` : `Term` (`suc` c), the topmost index in `Fin` (`suc` c). The existing substitution σ : `AList` $c\ k$ is lifted to `liftAList` σ : `AList` (`suc` c) (`suc` k) to accommodate the new variable.

Disjunction and Conjunction.

```
disj : Goal → Goal → Goal
disj g₁ g₂ s = interleave (g₁ s) (g₂ s)

conj : Goal → Goal → Goal
conj g₁ g₂ s = bindStep (g₁ s) g₂
```

`disj` expresses logical *or*: both goals are applied to the same input state `s`, yielding two independent streams of solutions, which are then merged with `interleave`. The choice of `interleave`, rather than plain concatenation, is what makes disjunction *fair*: if g_1 produces infinitely many solutions, concatenation would never reach any solution from g_2, whereas `interleave` alternates between the two streams so that every solution in either branch is eventually reached.

`conj` expresses logical *and*: g_1 is applied to `s` first, producing a stream of states that each satisfy g_1; then g_2 is applied to every state in that stream and the results are merged. This is precisely the monadic bind of the stream type, so `bindStep` is the right primitive: it consumes the output of g_1 one element at a time, applies g_2 to each, and interleaves the resulting sub-streams via the round-robin queue in `bindDriver`, ensuring that conjuncts are searched in a depth-first-with-interleaving order rather than pure depth-first, which could loop on an infinite branch.

Together, `disj` and `conj` reduce the entire search strategy to two primitives: `interleave` handles branching, and `bindStep` handles sequencing. Every other search combinator in microKanren is a derived form of these two.

3.5 Run Interface

The `run` function executes a goal program against a fresh variable and collects the first n answers, using a *fuel* counter f to limit `later` steps and prevent divergence:

```
takeListS : ℕ → ℕ → Step → List State
takeListS _ _ fail = []
takeListS f zero (yield _ _) = []
takeListS f (suc n) (yield s xs′)
    = s ∷ takeListS f n (xs′ .step)
takeListS zero n (later _) = []
takeListS (suc f) n (later xs′)
    = takeListS f n (xs′ .step)

takeList : ℕ → ℕ → KStream → List State
takeList f n xs = takeListS f n (xs .step)
```

`takeList f n xs` forces one `.step` observation and delegates to `takeListS`, which collects at most `n` answer states spending at most `f` steps of fuel on `later` constructors. The five clauses of `takeListS` reflect the two independent stopping conditions and the three `Step` shapes:

- *`fail`:* the stream is exhausted; return the empty list regardless of remaining fuel or answer quota.
- *`later`, fuel* $= 0$*:* the fuel counter has run out; return the empty list, treating this branch as timed out.
- *`later`, fuel* > 0*:* decrement the fuel counter and force one step via `xs′ .step`, continuing the traversal.
- *answer quota* $= 0$*:* the requested number of answers has been collected; stop immediately.
- *`yield s xs′`:* a result `s` is available; prepend it to the list and recurse on `xs′ .step`, decrementing the answer quota.

This design directly reflects the execution strategy of the search: because the stream may contain infinitely many `later` suspensions interspersed with `yield`ded answers, a consequence of `interleave` and `bindStep` inserting `later` constructors to maintain guardedness, `takeList` must be prepared to traverse an unbounded number of `later` steps between consecutive answers. The fuel counter makes this traversal total in Agda: it converts a potentially nonterminating coinductive traversal into a structurally recursive function on the fuel argument, at the cost of returning fewer than `n` answers if the fuel runs out before the quota is met. In practice the caller supplies a large fuel value so that the limit is never reached for finite searches.

The single-variable interface. For programs with a single logic variable, the library exposes:

```
run : ℕ → (∀ {c} → Term (suc c) → Goal) → List (∃ Term)
run n f = map (λ s → reify s (var (fromℕ 0)))
              (takeList 100000 n (callFresh f emptyState))
```

`run n f` introduces one fresh variable, applies `f` to it via function `callFresh`, then collects up to `n` results by reifying the variable at index 0_F in each answer state. For programs with *multiple* logic variables the recommended pattern is to pre-allocate a state with the desired variable count and pass the corresponding `var (# i)` terms directly, as illustrated in the case study (Section 4).

4 Case Study: The High-Five Puzzle

To validate the formalization beyond its own correctness theorems, we encode the *high-five puzzle*, a five-house logic puzzle originally defined in [15]. The puzzle assigns a company, a department, and an employee to each of five office positions subject to eleven relational constraints.

Puzzle statement. The setting is an office building whose 41st through 45th floors each house exactly one business, one business type (department), and one business owner (employee). The goal is to determine which combination of company, department, and employee occupies each floor from the following eleven clues:

(1) One business is Watershed Co.
(2) One business type is a real-estate business.
(3) Edwina's company is exactly one floor above Nelson & Leopold.
(4) Edwina's company is exactly one floor below the accounting firm.
(5) Brierwood Ltd. is on the 44th floor.
(6) Zed's company is exactly one floor above Ogden's.
(7) Keith's business is on the 45th floor.
(8) Trish works at the public-relations firm.
(9) Trish's firm is exactly two floors above Glyptic Co.
(10) Glyptic Co. and the Thebes Group are the literary agency and the web-design firm, in some order.
(11) The web-design firm is on the 41st or 42nd floor.

Clues 3–4 and 6 express positional constraints between floors; clues 5, 7, and 11 fix or restrict individual floor assignments; the remaining clues establish membership of a value in one of the five floor slots without specifying the exact position. Together they admit a unique solution (Section 4.4).

4.1 Domain Type And Module Instantiation

We instantiate `MicroKanren` with a concrete atom type:

```
data Atom : Set where
  Brierwood GlypticCo WatershedCo ThebesGroup : Atom
  NelsonLeopold RealEstate PublicRelations : Atom
  WebDesign Literary Accounting Keith : Atom
  Trish Edwina Ogden Zed nil : Atom

Atom-≟ : DecidableEquality Atom
-- 256 cases by exhaustive pattern match on all
-- constructor pairs

open import Main        Atom Atom-≟
open import MicroKanren Atom Atom-≟
```

4.2 Encoding The Puzzle

We declare the total variable count once:

```
PuzzleN : ℕ
PuzzleN = 15  -- 3 variables per house × 5 houses
```

Each floor $i \in \{1, \ldots, 5\}$ (corresponding to floors 41–45) is represented by three logic variables: company, department, and employee. All 15 variables share the type `Term PuzzleN`, which is essential: every $\equiv_k$ goal checks at runtime that the state's scope counter equals the type index of its arguments, so all variables must carry the same index.

Variables are pre-allocated by starting the search from a state with `PuzzleN` free variables, and naming them via `var (#i)` wrapped in house-triples:

```
H = Term PuzzleN × Term PuzzleN × Term PuzzleN

h₁ = (var (# 0)  , var (# 1)  , var (# 2))
h₂ = (var (# 3)  , var (# 4)  , var (# 5))
...
h₅ = (var (# 12) , var (# 13) , var (# 14))

puzzleState : State
puzzleState = mkState (nil {n = PuzzleN})
```

The `Atom` values (companies, departments, employees) are lifted into `Term PuzzleN` with the `val` constructor. Positional constraints are not expressed with arithmetic on floor numbers; instead they are unrolled into a finite disjunction of conjunctions over consecutive floor variables. For example, "Edwina's company is exactly one floor above Nelson & Leopold" (clue 3) expands to a four-way disjunction over adjacent house pairs:

```
nelsonleopold-left-edwina : Goal
nelsonleopold-left-edwina
   = someAdj λ (c , _ , _) (_ , _ , e) →
          conj (c ≡ₖ val (company NelsonLeopold))
               (e ≡ₖ val (employee Edwina))
```

where `someAdj` ranges over the four adjacent pairs (h_1, h_2), ..., (h_4, h_5). The two-floor gap in clue 9 is handled analogously with a helper ranging over the three pairs at distance two. All eleven constraints are composed with `conj*` into a single `puzzle : Goal`.

Nested `callFresh` and the `callFresh2` Combinator. The micro-Kanren idiom for introducing multiple fresh logic variables is to nest `callFresh` calls. In our formalization this pattern is subtly broken when both variables appear in a single $\equiv_k$ goal. When the outer `callFresh` evaluates on a state with counter c_0, it introduces x : `Term` (`suc` c_0). The inner `callFresh` extends scope again, so y : `Term` (`suc` (`suc` c_0)). Because $\equiv_k$ checks at runtime that the state counter matches the term's scope index, x is now at the wrong scope and the goal silently returns `mzero`.

We resolve this with `callFresh2`, which allocates both variables in the same extended scope by applying `liftAList` twice before invoking the continuation:

```
callFresh2 : (∀ {c} → Term (suc (suc c)) →
                      Term (suc (suc c)) →
                      Goal) → Goal
callFresh2 f (mkState {c} σ) =
  f {c} (var Fin.zero) (var (Fin.suc Fin.zero))
        (mkState (liftAList (liftAList σ)))
```

After two applications of `liftAList`, the original variables at indices $0, \ldots, c-1$ are shifted to $2, \ldots, c+1$. The two fresh variables occupy indices 0 and 1, both unbound in the extended substitution, and both typed at `Term` (`suc` (`suc` c)). The continuation receives them as a pair in identical scope, so $\equiv_k$ goals comparing either variable to a value work without weakening. Pre-allocating all variables from a single `mkState (nil {n = PuzzleN})` remains the preferred approach when more than two fresh variables are needed.

4.3 Translating Constraints

Each Racket constraint maps directly to a combination of `disj` and `conj`. The helper `some` ranges over all five houses: `some g = disj* (map g houses)`. For example, "some house has company Watershed" becomes:

```
watershed : Goal
watershed = some λ (c , _ , _) →
               c ≡ₖ val (company WatershedCo)
```

Multi-field membership requires both fields to belong to the same house, so each disjunct is a `conj` of two equalities over the same house's variables:

```
trish-pr : Goal
trish-pr = some λ (_ , d , e) →
   conj (d ≡ₖ val (department PublicRelations))
        (e ≡ₖ val (employee Trish))
```

All eleven constraints are composed with `conj*`. The wildcards in the Racket patterns disappear entirely: since unmentioned slots remain unconstrained, no fresh anonymous variables are needed.

4.4 Evaluation And Results

```
solution : List (∃ Term)
solution = runPuzzle 1
  where
     states = takeList 100000 n (puzzle puzzleState)
     runPuzzle n = map (λ s → reify s houseList) states
```

Evaluated via the GHC backend, the program terminates immediately and yields the unique solution:

Table 1: Evaluation results for the puzzle.

House	Company	Department	Employee
H1	Glyptic	Web-Design	Ogden
H2	Thebes-Group	Literary	Zed
H3	Nelson-&-Leopold	Public-Relations	Trish
H4	Brierwood	Real-Estate	Edwina
H5	Watershed	Accounting	Keith

This matches the output of the original Racket miniKanren program, confirming that the Agda formalization faithfully implements the search semantics.

5 Case Study: Infinite Enumeration

The High-Five puzzle has a finite search space, so it does not exercise the `later`-based fairness of the coinductive stream model. To validate the coinductive machinery on an infinite domain, we define a goal that enumerates all Peano naturals without any pragma.

We represent Peano numerals as atoms and instantiate the engine over a dedicated atom type:

```
data NatAtom : Set where
  O : NatAtom
  S : NatAtom → NatAtom
```

The goal `natsFrom` is defined by copattern matching on the `.step` field of the coinductive `KStream`. Since every corecursive call appears directly under `yield` or `later` in a `.step` clause, Agda's guardedness checker accepts the definition under the `--safe` flag:

```
natsFrom : NatAtom → ∀ {c} → Term (suc c) → Goal
natsFrom n x s .step with (x ≡ₖ val n) s .step
... | fail       = later (natsFrom (S n) x s)
... | yield s' _ = yield s' (natsFrom (S n) x s)
... | later _    = later (natsFrom (S n) x s)
```

The `fail` branch advances the counter and emits a `later` suspension rather than discarding the remaining search. The `yield` branch records the current solution and also continues with the next natural, so the stream is infinite. We retrieve the first k answers with the standard run interface:

```
firstN : ℕ → List (∃ Term)
firstN k = run k (natsFrom O)
```

Evaluating `firstN 4` produces $[\mathtt{O}, \mathtt{S\,O}, \mathtt{S(S\,O)}, \mathtt{S(S(S\,O))}]$, confirming that `takeList`'s fuel parameter correctly bounds coinductive unfolding even when the stream never exhausts.

6 Discussion

Interleave. `interleave` is a single corecursive function defined by a copattern equation `interleave xs ys .step with xs .step`. Every recursive call appears directly under a `Step` constructor (`yield` or `later`), which is one guarded step from the copattern boundary. Agda's checker accepts the definition without any pragma.

The guardedness problem for `bind`. The central difficulty is that fair binding must apply a goal to every state in a stream and interleave the resulting sub-streams with the remainder: a pattern where the recursive call must be passed as an argument to another function. The guardedness checker accepts corecursive calls only when they appear directly under a constructor, never under an arbitrary function call; it does not look through function bodies.

Resolution via a round-robin queue. We resolve this issue using `bindDriver`: the recursive call to `bindDriver` always appears as the direct argument of `later` or `yield` in the body of a copattern `.step` equation. There is no intermediate function application wrapping the recursive call. Agda's guardedness checker therefore accepts all nine dispatch cases without any pragma, and the formalization type-checks under the `--safe` flag, ruling out postulates, axioms, and `TERMINATING` annotations globally.

Algebraic properties of `interleave`. We prove two membership-equivalence lemmas for the interleaving combinator. The first establishes that $s \in$ `interleave mzero` $ys \Leftrightarrow s \in ys$, which follows because `mzero .step = fail` makes the two streams definitionally equal. The other establishes that $s \in$ `interleave` xs `mzero` $\Leftrightarrow s \in xs$, proved by inverting membership with `∈-interleave⁻` and eliminating the right branch with `¬∈-fail`. Together with the existing `disj-sound` and `disj-complete-l/r` results, these lemmas show that `disj` acts as a set-union operator with `const mzero` as the identity element, a key algebraic property of the search combinator even in the absence of a full denotational semantics.

7 Related Work

Verified Logic Programming Engines. Jaume [9] presents a full formalization of SLD-resolution in the Calculus of Inductive Constructions, verified with Coq. The formalization covers first-order term unification (derived from an earlier proof for quasi-terms), explicit renaming in SLD-derivations, and the switching and lifting

lemmas needed to establish soundness and completeness of Prolog with respect to its denotational semantics. Our work differs from Jaume's in three important ways. First, we target microKanren's *fair-interleaving* search rather than Prolog's depth-first SLD resolution; the two search strategies differ fundamentally in completeness guarantees for infinite search spaces. Second, our term language is scope-indexed: variables are `Fin` values and substitutions are `AList m n` terms, so scope violations are type errors; Jaume's formalization uses untyped terms and separate well-formedness conditions. Third, we require no coinduction because Jaume's SLD derivations are finite trees, whereas our coinductive stream model is essential for representing the potentially infinite answer sequences of fair search.

miniKanren Semantics. Rozplokhas et al. [16] give two certified formal semantics for the core miniKanren language, formalized in Coq. The denotational semantics interprets goals as sets of substitutions corresponding to the minimal Herbrand model for definite logic programs; the operational semantics models the actual search procedure as a labeled transition system with interleaving. They prove soundness and completeness of the operational semantics with respect to the denotational one, and the Coq development is publicly available. Crucially, their language includes recursive relational definitions (the `Invoke` constructor), which allows programs such as `append`^{o} to be expressed directly; our formalization covers only the four microKanren primitives and treats recursion as a meta-level concern. A further difference is in the treatment of variables: Rozplokhas et al. use a dynamically scoped variable counter with untyped terms, whereas our `Fin`-indexed terms and `AList m n` substitutions make scope safety a static guarantee. Finally, their soundness and completeness results concern the *search process* as a whole; we instead focus on a machine-verified correctness certificate for the *unification primitive*, additionally establishing maximality of the computed unifier, a property not addressed in their work.

Unification in Type Theory. McBride [12] shows that first-order unification can be made structurally recursive by encoding the variable set as a dependent index: as each variable is eliminated by substitution, the index decreases, and termination follows from the type structure without a separate measure argument. The algorithm and its partial correctness proof were verified in the LEGO proof assistant. Our formalization builds directly on McBride's insight, re-implementing his algorithm in Agda with two significant extensions: (1) we use accumulating association-list substitutions (`AList m n`) rather than functions on variable sets, following a design direction McBride himself suggested; and (2) we package soundness, completeness, and maximality into the `AmguResult` type, so the correctness certificate is an intrinsic part of the algorithm's return type rather than a separately proved relation. We then connect this verified unifier directly to microKanren's search engine, which McBride's paper does not address.

Allais et al. [2] develop a *generic* universe of syntaxes with binding in Agda, in which renaming, substitution, and other scope-safe traversals are implemented once for every syntax in the universe and their properties are derived automatically. Variables are represented with de Bruijn indices and scopes are tracked as type indices, the same representation we use. The key difference in scope is that their framework is parameterized over an arbitrary syntax description, whereas our formalization is specialized to first-order terms over a decidable atom type with a fixed arity of zero. Their generic traversal machinery is not needed here because the only operations on terms required by microKanren are unification and `walk`-based dereferencing, both of which are specific enough to be defined directly. The two developments share the philosophical commitment to making ill-scoped terms unrepresentable by construction.

Proof Search in Agda. Kokke and Swierstra [10] implement a Prolog-style resolution procedure directly inside Agda, integrating it with Agda's reflection mechanism to produce a first-class proof-search tactic (`auto`). Their interpreter is written in the safe fragment of Agda (no postulates; all definitions pass the termination checker) and uses coinduction to handle the fact that resolving a Prolog query may not terminate, mirroring our use of coinductive streams to model non-terminating relational search. The key difference in scope is that their search targets *proof construction*: given a hint database of lemmas, it synthesises a proof term witnessing a given type. Our work, by contrast, targets *answer enumeration*: given a microKanren goal, it produces a stream of substitutions representing all satisfying assignments. Both developments share the architectural choice of representing open-ended search coinductively to keep all code total, and both operate within Agda's type system to obtain static guarantees not achievable in a dynamically typed host language.

8 Conclusion

We have presented a mechanized formalization of microKanren in Agda, covering scope-safe terms with decidable equality, a machine-verified accumulating most-general unifier, a coinductive stream type for non-deterministic search, and the four microKanren primitives. The formalization establishes soundness, completeness, and maximality of the unification goal, and type-checks entirely under the `--safe` flag.

Future work includes proving a full denotational semantics for the four primitives, including associativity of `bindStep` and commutativity of `interleave`, and extending the formalization to tabling [4] and constraint domains [3, 8].

Artifact Availability

The Agda source code for the formalization is available at:

https://github.com/lives-group/agda-kanren

The development type-checks under the `--safe` flag in Agda 2.7.0.1 with standard library 2.2.

Use of Generative AI

Generative AI technologies (specifically Gemini) were used in the preparation of this work for grammatical review. The authors reviewed all AI-generated suggestions and take full responsibility for the content and validity of the final text.

References

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming Infinite Structures by Observations. In *Proceedings of the*

40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13). ACM, 27–38. doi:10.1145/2429069.2429075

[2] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2021. A Type and Scope Safe Universe of Syntaxes with Binding: Their Semantics and Proofs. *Journal of Functional Programming* 31 (2021), e22. doi:10.1017/S0956796820000076

[3] Claire E. Alvis, Jeremiah J. Willcock, Kyle M. Carter, William E. Byrd, and Daniel P. Friedman. 2011. cKanren: miniKanren with Constraints. In *Proceedings of the 2011 Workshop on Scheme and Functional Programming (Scheme '11)*. https://www.schemeworkshop.org/2011/papers/Alvis2011.pdf

[4] William E. Byrd. 2009. *Relational Programming in miniKanren: Techniques, Applications, and Implementations*. Ph. D. Dissertation. Indiana University. https://scholarworks.iu.edu/dspace/handle/2022/8777

[5] Venanzio Capretta. 2005. General Recursion via Coinductive Types. *Logical Methods in Computer Science* 1, 2 (2005). doi:10.2168/LMCS-1(2:1)2005

[6] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The Reasoned Schemer*. The MIT Press. 176 pages.

[7] Jason Hemann and Daniel P. Friedman. 2013. μKanren: A Minimal Functional Core for Relational Programming. In *Proceedings of the 2013 Workshop on Scheme and Functional Programming (Scheme '13)*.

[8] Jason Hemann and Daniel P. Friedman. 2017. A Framework for Extending microKanren with Constraints. In *Technical Communications of the 29th and 30th Workshops on (Constraint) Logic Programming and 24th International Workshop on Functional and (Constraint) Logic Programming (Electronic Proceedings in Theoretical Computer Science, Vol. 234)*. 135–149. doi:10.4204/EPTCS.234.10

[9] Mathieu Jaume. 1999. A Full Formalization of SLD-Resolution in the Calculus of Inductive Constructions. *Journal of Automated Reasoning* 23, 3–4 (1999), 347–371. doi:10.1023/A:1006242018697

[10] Pepijn Kokke and Wouter Swierstra. 2015. Auto in Agda: Programming Proof Search Using Reflection. In *Mathematics of Program Construction — 12th International Conference, MPC 2015 (Lecture Notes in Computer Science, Vol. 9129)*. Springer, 205–227. doi:10.1007/978-3-319-19797-5_10

[11] Per Martin-Löf. 1984. *Intuitionistic Type Theory*. Bibliopolis. Notes by Giovanni Sambin of a series of lectures given in Padova , June 1980.

[12] Conor McBride. 2003. First-Order Unification by Structural Recursion. *Journal of Functional Programming* 13, 6 (2003), 1061–1075. doi:10.1017/S0956796803004957

[13] Ulf Norell. 2007. *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph. D. Dissertation. Chalmers University of Technology.

[14] Ulf Norell. 2009. Dependently Typed Programming in Agda. In *Advanced Functional Programming*. Lecture Notes in Computer Science, Vol. 5832. Springer, 230–266. doi:10.1007/978-3-642-04652-0_5

[15] Penny Press / Dell Magazines. 2016. Original Logic Problems. Penny Press / Dell Magazines, New York, NY. Recurring magazine; 128 pages, 90 logic problems per issue. https://www.pennydellpuzzles.com/products/logic-math/original-logic-problems/.

[16] Dmitry Rozplokhas, Andrey Vyatkin, and Dmitry Boulytchev. 2020. Certified Semantics for Relational Programming. In *Programming Languages and Systems: 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 – December 2, 2020, Proceedings* (Fukuoka, Japan). Springer-Verlag, Berlin, Heidelberg, 167–185. doi:10.1007/978-3-030-64437-6_9

[17] Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2024. Programming Language Foundations in Agda. https://plfa.github.io