Typed Embedding of miniKanren for Functional Conversion

IGOR ENGEL, JetBrains Research, Germany and Constructor University, Germany
EKATERINA VERBITSKAIA, JetBrains Research, Netherlands and Constructor University, Germany

Relational programming enables program synthesis through a verifier-to-solver approach. An earlier paper introduced a functional conversion that mitigated some of the inherent performance overhead. However, the conversion was inelegant: it was oblivious to types, demanded determinism annotations, and implicit generator threading. In this paper, we address these issues by providing a typed tagless-final embedding of miniKanren into Haskell. This improvement significantly reduces boilerplate while preserving, and sometimes enhancing, earlier speedups.

Additional Key Words and Phrases: Relational Programming, Functional Conversion, Typed Tagless Final Embedding

ACM Reference Format:
Igor Engel and Ekaterina Verbitskaia. 2025. Typed Embedding of miniKanren for Functional Conversion. 1, 1 (October 2025), 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The innate ability of a relational program to run in multiple modes reduces the complicated task of *finding* a solution to the much simpler task of *checking* that a candidate satisfies the specification [4]. For example, a program which verifies whether a given sequence of vertices forms a path in a graph can enumerate paths. Likewise, a relation that checks whether some variable assignment satisfies a propositional formula can be run in reverse to search for satisfying assignments. A notable instance of this verifier-to-solver approach enables program synthesis: by running a relational interpreter backwards, we can generate programs whose evaluation yields a given value.

One downside to this approach is its high overhead resulting in low performance. A relational interpreter should be carefully crafted to synthesize non-trivial programs within reasonable time constraints. Moreover, the implementation of miniKanren itself has to be heavily optimized which undermines its reputation for being easy to embed in any general-purpose language.

The miniKanren community has devoted a lot of effort over the years to make program synthesis a reality. Among such efforts is the functional conversion [9] whose purpose is to translate a relation with a given direction into a functional program. As a result, much of the overhead of relational programming is neutralized, especially if paired with advanced program transformation techniques such as specialization [8].

While the implementation of the functional conversion described in [9] improves execution time, it exhibits several limitations. Firstly, because it relies on an untyped deep embedding of miniKanren in Haskell, it fails to take advantage of the Haskell type system, which leads to a loss of static guarantees as well as forces the user to painstakingly thread generator parameters.

Authors' Contact Information: Igor Engel, igorengel@mail.ru, JetBrains Research, Munich, Germany and Constructor University, Bremen, Germany; Ekaterina Verbitskaia, ekaterina.verbitskaya@jetbrains.com, JetBrains Research, Amsterdam, Netherlands and Constructor University, Bremen, Germany.

ACM XXXX-XXXX/2025/10-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Secondly, the absence of automatic determinism analysis leaves some optimization opportunities untapped. Finally, the deep embedded representation hinders composition and extension—qualities that are among miniKanren's greater strengths.

In this paper, we describe a typed, tagless-final [1] embedding of miniKanren in Haskell. Along with automatic determinism analysis, it refines the earlier functional conversion, enhancing both performance and developer experience. In addition to this, the embedding opens the door to composable and extensible miniKanren implementations in statically typed functional languages.

2 Background

In this section we introduce the background necessary to understand the contributions of this paper.

2.1 miniKanren

The miniKanren family of programming languages is created to be simple to understand, implement, and extend with new features.[1] One distinguishing characteristic of miniKanren is its complete search enabled by design. Any existing solution to a query will be found eventually [6] which makes miniKanren uniquely suitable for declarative setting and reverse execution.

Most miniKanren dialects are implemented as a set of combinators for basic operations such as a unification or conjunction and disjunction. However, program transformation is complicated for shallow embeddings in languages featuring referential transparency. Primarily due to this fact the proof-of-concept functional conversion was implemented for a deeply embedded miniKanren and manipulated abstract syntax trees explicitly. The syntax in Fig. 1 describes the core language used in this paper, also known as microKanren [2]. We also assume inverse η-delay to accompany every relation invocation and not used elsewhere.

C	$=$	$\{C_i^{k_i}\}$	constructors with arities
$\mathcal{R}$	$=$	$\{R_i^{k_i}\}$	relational symbols with arities
$\mathcal{T}_V$	$=$	$V \cup \{C_i\,(t_1, \ldots, t_{k_i}) \mid t_j \in \mathcal{T}_V\}$	terms on set of variables V
$\mathcal{G}_V$	$=$	$\mathcal{T}_V \equiv \mathcal{T}_V$	unification
	$\mid$	$\mathcal{G}_V \wedge \mathcal{G}_V$	conjunction
	$\mid$	$\mathcal{G}_V \vee \mathcal{G}_V$	disjunction
	$\mid$	$R_i^{k_i}\,(t_1, \ldots, t_k), t_j \in \mathcal{T}_V$	relational symbol invocation
	$\mid$	$\mathbf{Fresh}_x\,\mathcal{G}_V$	fresh variable introduction
$\mathcal{D}_V$	$=$	$R_i\,(x_1, \ldots, x_{k_i}) \equiv \mathcal{G}_V, x_j \in V$	relational symbol definition
$\mathcal{P}$	$=$	$\mathcal{D}_V^*$	miniKanren program

Fig. 1. miniKanren syntax

The addition relation written in this syntax takes the following shape, where R_{add} is the only relational symbol, C_0 and C_S are constructors, and x, x', y, z, z' are variables:

$$R_{\text{add}}\,(x, y, z) \equiv (x \equiv C_0 \wedge y \equiv z) \vee (\mathbf{Fresh}_{x'}\,\mathbf{Fresh}_{z'}\,(x \equiv C_S\,(x') \wedge R_{add}\,(x', y, z') \wedge z \equiv C_S\,(z')))$$

[1] Website of the miniKanren programming languages family: http://minikanren.org/

2.2 Normal Form

Many program transformations become simpler when done over a normalized representation, such as the superhomogenous normal form used in the Mercury programming language [7]. It is in essence a disjunctive normal form with unique variables, and with extra care taken not to allow exponential explosion in the number of clauses. To avoid it, we create a new relation whenever a disjunction takes place within a conjunction. The process of normalization is described in detail in [9], here we only present the normal form syntax, see Figure 2. Notice that there are also no **Fresh** declarations in this form, and each element of V is assumed to refer to the same variable within the invocation.

$$
\begin{array}{lcll}
\mathcal{FT}_V & = & V \cup \{C_i\left(x_1, \ldots, x_{k_i}\right) \mid x_j \in V, a \neq b \implies x_a \neq x_b\} & \text{linear flat terms} \\
\mathbf{Base}_V & = & V \equiv \mathcal{FT}_V & \text{linear flat unification} \\
 & \mid & R_i\left(x_1, \ldots, x_{k_i}\right), x_j \in V, a \neq b \implies x_b \neq x_b & \text{linear flat call} \\
\mathbf{Conj}_V & = & \bigwedge\left(g_1, \ldots, g_n\right), g_i \in \mathbf{Base}_V & \text{normal conjunction} \\
\mathcal{K}_V^N & = & \bigvee\left(c_1, \ldots, c_n\right), c_i \in \mathbf{Conj}_V & \text{normal form}
\end{array}
$$

Fig. 2. Normalized miniKanren syntax

2.3 Modes

A relational program can be executed in multiple directions, also called *modes*. Mode systems assign each variable in a relation a mode annotation that denotes how the variable is used in it. A logic/functional programming language Mercury features an advanced mode system that informs its compiler on possible optimizations. The functional conversion of miniKanren relies on the simplest mode system with only two possible annotations: **In** and **Out**. **In** denotes a variable whose value is fully known when the operation is executed. **Out** is reserved for a variable with unknown value which is going to be bound after the operation is executed.

When all variables of a relation are annotated with modes, we say that the relation has a mode. We use the term *direction* to describe a relation call whose arguments are annotated. For example, consider the following directions of the addition relation:

$$
\begin{gathered}
R_{\text{add}}\left(x^{\text{In}}, y^{\text{In}}, z^{\text{Out}}\right) \equiv \left(x^{\text{In}} \equiv C_0 \wedge y^{\text{In}} \equiv z^{\text{Out}}\right) \\
\vee \left(\mathbf{Fresh}_{x'}\, \mathbf{Fresh}_{z'}\left(x^{\text{In}} \equiv C_S\left(x'^{\text{Out}}\right) \wedge R_{add}\left(x'^{\text{In}}, y^{\text{In}}, z'^{\text{Out}}\right) \wedge z^{\text{Out}} \equiv C_S\left(z'^{\text{In}}\right)\right)\right) \\
R_{\text{add}}\left(x^{\text{Out}}, y^{\text{Out}}, z^{\text{In}}\right) \equiv \left(x^{\text{Out}} \equiv C_0 \wedge y^{\text{Out}} \equiv z^{\text{In}}\right) \\
\vee \left(\mathbf{Fresh}_{x'}\, \mathbf{Fresh}_{z'}\left(x^{\text{Out}} \equiv C_S\left(x'^{\text{In}}\right) \wedge R_{add}\left(x'^{\text{Out}}, y^{\text{Out}}, z'^{\text{In}}\right) \wedge z^{\text{In}} \equiv C_S\left(z'^{\text{Out}}\right)\right)\right) \\
R_{\text{add}}\left(x^{\text{In}}, y^{\text{Out}}, z^{\text{Out}}\right) \equiv \left(x^{\text{In}} \equiv C_0 \wedge y^{\text{Out}} \equiv z^{\text{Out}}\right) \\
\vee \left(\mathbf{Fresh}_{x'}\, \mathbf{Fresh}_{z'}\left(x^{\text{In}} \equiv C_S\left(x'^{\text{Out}}\right) \wedge R_{add}\left(x'^{\text{In}}, y^{\text{Out}}, z'^{\text{Out}}\right) \wedge z^{\text{Out}} \equiv C_S\left(z'^{\text{In}}\right)\right)\right)
\end{gathered}
$$

In a well-moded relation, data flows from **In** to **Out**, and once a variable's value is known, it is never forgotten. This means that a variable with the **In** mode can never become **Out**, while **Out** becomes **In** after the current operation is executed. This intuition translates well to functional code, where each variable is expected to be bound by some value before use. The functional conversion employs a mode analysis capable of inferring mode annotations given a direction. The mode analysis algorithm does not fall in the scope of this paper; thus, the reader is referred to [9] for a

detailed description of the approach. Its details are not essential for understanding the contributions presented here.

2.4 Functional Conversion

In this section we briefly summarize the functional conversion described in [9] which the reader can refer to for deeper comprehension. The functional conversion mirrors the relational conversion [5] aimed at generating relations from functions thus coming full circle in the verifier-to-solver approach.

Given a miniKanren relation with a concrete direction, the goal of a functional conversion is to construct a function which yields the same answers as the relation would. Because the search in miniKanren is complete, disjunctions and conjunctions can be reordered, which will hopefully result in faster execution. Mode analysis guides this reordering as well as classifies unifications into equality checks, assignments, pattern matches and generations. Consider the following functional counterpart of $R_{\text{add}}\left(x^{\mathbf{In}}, y^{\mathbf{Out}}, z^{\mathbf{Out}}\right)$ which computes such pairs (y, z) that $x + y = z$, where x is known.

```
1  data Term = Z
2            | S Term
3            deriving (Show, Eq)
4  addoIOO :: MonadPlus m => Term -> m Term -> m (Term, Term)
5  addoIOO x addoIOO_z =
6      msum [
7          do {
8              guard (x == Z);
9              (y, z) <- do { x0 <- addoIOO_z; return (x0, x0); }
10             return (y, z)
11         },
12         do {
13             S x' <- return x;
14             (y, z') <- delay (addoIOO x' addoIOO_z);
15             z <- return (S z');
16             return (y, z)
17         }
18     ]
```

The function `addoIOO` produces infinitely many pairs of Peano numbers; thus the output of the function should be a stream of `(Term, Term)`. The `Stream` is not always the most efficient data structure to represent the output, especially when the computation is deterministic. Because of this, the converter generates functions parameterized by an arbitrary **`Monad`** `m` that can be specified at the call site.

Let us now illustrate the conversion process by this example. Once mode analysis is finished, disjunctions are translated into **`msum`** of several monadic computations (line 6). Each such computation is produced from a conjunction of either calls or unifications. Note that moded unifications can become an equality test (line 8), a pattern matching (line 13), an assignment (line 15), or use term generation (line 9). Generation is required whenever both sides of a unification are annotated **Out**, producing a stream of all possible values.

A drawback of the previous implementation is the need to supply generators as explicit function arguments. Moreover, when several variables are generated, a separate generator must be provided for each one—even if they share the same type and could be produced in exactly the same way. This requirement makes the resulting code inelegant, as well as prone to user errors when they later have to supply the generators manually.

3 Typed Shallow Embedding

The functional conversion implemented in [9] operates on an untyped version of miniKanren even though the target languages are Haskell and OCaml. These languages feature expressive static type systems, and it would be a shame to forgo their benefits. Besides explicit passing of generators this approach also erases all information about the type of specific variables, and combine all constructors in one type called `Term`. Figure 3 shows an example of such type, which integrates natural numbers, lists, and binary trees. This makes values such as `S (Cons (Leaf) (S Nil))` possible, even though they have no meaningful interpretation.

```
data Term
    = Cons Term Term
    | Leaf
    | Nil
    | Node Term Term Term
    | S Term
    | Z
```

Fig. 3. Generated *Term* type

To overcome these issues, we need a typed miniKanren embedding capable of linking each logical value to its ground counterpart. Existing candidates fall short. Haskell embedding typed-Kanren [3] supplies the required logical-type machinery but only for execution, not for inspecting or analysing relations. Simultaneously, OCanren offers a typed OCaml embedding yet targets a language less convenient for our conversion pipeline. We therefore develop a new, tagless-final [1] embedding in Haskell that lets programmers write type-safe and composable relations, provides the correspondence between logic and ground types, and remains open to future extensions.

The embedding consists of two interfaces expressed as Haskell typeclasses:

- `LogicType` that builds correspondence between the logical and the underlying type.
- `Kanren` that describes a way to interpret a miniKanren relation.

These typeclasses are then instantiated to provide concrete logic types and interpreters of relations.

3.1 LogicType

miniKanren interpreters do not operate on concrete Haskell types; instead they expect logical types whose structures mirror the original but contain placeholders for variables, also called *holes*. The `LogicType` typeclass with associated type `WithLogic` enforce the shape of a logical type and establish a correspondence between ground and logical types. To support multiple interpreters, logic types require variables to represent holes and have to be polymorphic over them, while a variable itself should be polymorphic over the type of its content. As such, we get the following definition of `LogicType` and `WithLogic`.

```
class LogicType a where
  data WithLogic a (var :: Type -> Type) :: Type
```

We also introduce two helper types: `Logic`, which represents a logical hole at top-level, and `Var`—a variable that contains a logical type.

```
data Logic a (var :: Type -> Type) = Free (Var a var) | Ground (WithLogic a var)
type Var x (var :: Type -> Type) = var (Logic x var)
```

To illustrate the interface, we will examine it step by step and implement it for a simple case of Peano numbers. Given the usual implementation of a natural number `Nat`, we can define the logic type in the following way.

```
data Nat = Z | S Nat deriving (Eq, Ord, Show)
instance LogicType Nat where
    data WithLogic Nat var = Z' | S' (Logic Nat var)
```

To map the underlying type to its logical counterpart, `LogicType` features functions `project` and `reify`.

```
  project :: a -> WithLogic a var
  reify :: WithLogic a var -> Maybe a
```

The function `project` maps the value of an underlying type to a fully-ground logical value of a logical type, while `reify` converts a logical value to its underlying value and returns **`Nothing`** if the input is not fully-ground. These functions should obey the following laws:

```
reify . project == Just
project . fromJust . reify == id -- if fromJust is defined
```

A simple instance for the Peano natural numbers might look like this:

```
    project Z = Z'
    project (S n) = S' $ Ground (project n)

    reify Z' = Just Z
    reify (S' (Ground n)) = S <$> reify n
    reify _ = Nothing
```

To be able to inspect logical values, we provide the following representations of term-style construction. It mirrors the typical structure of miniKanren values: a constructor with a fixed number of arguments of a logical type.

```
data Field a var where
  Field :: LogicType x => Logic x var -> Field a var

data Constructor a = Constructor
  { name :: String
  , construct :: forall var. [Field a var] -> WithLogic a var
  }
```

`Field` is a type-erasing GADT, which ensures arguments may have different types. `Constructor` represents a particular constructor of a term, with an explicit name and a way to create a value.

`LogicType` introduces the function `quote` which obeys the following law.

```
  quote :: WithLogic a var -> (Constructor a, [Field a var])
  uncurry construct . quote == id
```

Implementing `quote` can be done with helper functions `quote0`, `quote1`, and others. For type-preserving translator, it is important for the names passed to the constructors in `quote` to correspond to the constructor names of the underlying type, however it is not in general a requirement of the system.

```
    quote Z' = quote0 "Z" Z'
    quote (S' n) = quote1 "S" S' n
```

Finally, it should be possible to unify logical types; thus `LogicType` contains the function `unifyVal`.

```
  unifyVal :: (Alternative rel) =>
              (forall t. LogicType t => Logic t var -> Logic t var -> rel ())
              -> WithLogic a var -> WithLogic a var -> rel ()
```

In addition to values to be unified, this function also accepts a *unification provider*. Unification provider is supplied by the interpreter (see 3.5) and conveys the meaning of the unification in interpreter's domain.

This function can be implemented for Peano numbers in the following way.

```
unifyVal _ Z' Z' = pure ()
unifyVal unif (S' x) (S' y) = unif x y
unifyVal _ _ _ = empty
```

The next function of `LogicType` are `derefVal` that is similar to `reify`, but works on values that can contain variables. To achieve this, it requires an environment that provides values for such variables through the first argument.

```
derefVal :: (Alternative rel) =>
            (forall t. LogicType t => Logic t var -> rel t)
            -> WithLogic a var -> rel a
```

Its implementation for natural numbers is as follows.

```
derefVal _ Z' = pure Z
derefVal deref (S' n) = S <$> (deref n)
```

The goal of the last function, `generate`, is to systematically enumerate all possible underlying values. It replaces the user-provided generators required in the untyped version 4.

```
generate :: Stream a
```

For natural numbers, the generator is rather simple:

```
generate = pure Z <> (S <$> generate)
```

Functions `unifyVal`, `derefVal`, and `generate` are redundant in the typeclass and can be implemented only by using `quote`. Such an implementation is provided, however its performance may be insufficient for particular interpreters. Because of that, the functions are included in the interface and can have custom implementations.

3.2 Kanren

The tagless-final approach represent a domain-specific language by an interface of polymorphic combinators. Their behavior is determined by interpreters—instances of the typeclass—rather than a fixed syntax tree resulting in extensible, type-safe embeddings.

In our implementation such interface is called `Kanren`. A type that has a `Kanren` instance denotes *relations that return a value*. Therefore, `Kanren rel` means that `rel Int` is a relation that produces a value of type `Int`.

Conjunctions are often expressed with a monadic bind, but bind enforces a fixed evaluation order—preventing the reordering of conjuncts that our functional conversion needs. Instead, we require the `Applicative` constraint which imposes no order. With this constraint it is impossible to use the value of an earlier conjunct in the following code; thus, for pure miniKanren relations, the type of the return value is fixed to unit `()`. The purpose of this parameter is described in 3.3.

Disjunction, by contrast, naturally fits the `Alternative` interface: `empty` represents failure, and `<|>` models branching without ordering constraints. Thus, we simply require `Alternative`, which matches the semantics of miniKanren choice exactly.

Finally, each `Kanren` instance has to specify a `KVar` type — a variable associated with this relation. It does not need to contain a value, but it has to be functorial over the type it is designated to contain. Thus, we have the following definition with basic miniKanren operations:

```
class (Alternative rel, Functor (KVar rel)) => Kanren rel where
    data KVar rel :: Type -> Type
    unify :: (LogicType a) => Logic a (KVar rel) -> Logic a (KVar rel) -> rel ()
```

```
    call_ :: (String, rel ()) -> rel ()
```

In our embedding we need two ways to introduce variables. The first one, which corresponds to **Fresh**$_v$, introduces an empty variable. The other denotes an argument of a relation that will, upon introduction, be bound to arguments passed to the call. It is equivalent to inserting a unification after **Fresh**$_v$, but is immediately visible whenever transformations of the relation are done.

```
data FreshType (rel :: Type -> Type) (t :: Type) where
    FreshVar :: FreshType rel t
    ArgVar :: (Kanren rel, LogicType t) => Logic t (KVar rel) -> FreshType rel t
```

The function `fresh` therefore receives a description of the variable to create and a continuation that builds the rest of the relation using that variable, and returns the fully assembled relation:

```
    fresh_ :: (LogicType t) => FreshType rel t -> (Var t (KVar rel) -> rel a) -> rel a
```

A handful of helpers and the `ApplicativeDo` Haskell extension allows writing relations concisely:

```
add :: (Kanren rel) => Logic Nat (KVar rel) -> Logic Nat (KVar rel) -> L Nat (KVar rel) -> (String, rel ())
add = relation3 "add" $ \x y z -> conde
    [ do
        x <=> zro -- infix synonym for unify
        y <=> z
        pure () -- ApplicativeDo requires that the final statement be exactly "pure E"
    , fresh2 $ \x' z' -> do
        x <=> suc x'
        z <=> suc z'
        call $ add x' y z'
        pure ()
    ]
```

3.3 KanrenEval

Although the `Kanren` typeclass makes it possible to define and interpret relations, it offers no mechanism to extract the result of relation execution. This may seem counterintuitive, but some interpreters, such as the normalizing interpreter, do not operate with the output, instead transforming the relation itself.

Nevertheless, it is an important property for evaluators which is why we extend `Kanren` with the `KanrenEval` typeclass. `KanrenEval` allows dereferencing variables which makes their value accessible as relation's return value. An interpreter can then define how to extract the resulting value (see 3.5).

```
class (Kanren rel) => KanrenEval rel where
    derefVar :: (LogicType a) => Var a (KVar rel) -> rel a
```

Combined with `derefVal` function from `LogicType` (3.1), the complete evaluation function is straightforward:

```
eval :: (KanrenEval rel, LogicType a) => Logic a (KVar rel) -> rel a
eval (Free v) = derefVar v
eval (Ground x) = derefVal eval x
```

3.4 Automatic Derivation of LogicType Instances

`LogicType` instances are highly constrained because of the requirement that the logic type mirrors miniKanren terms and remains interoperable with the underlying type. Only algebraic data types (and GADTs) qualify, therefore exactly one valid implementation of `LogicType` is admitted per underlying type—a fact we exploit for automatic derivation.

`TemplateHaskell` is capable of generating this unique instance for a given algebraic data type. For performance reasons (see Section 5) the implementations of functions in this typeclass must be annotated `INLINABLE`. This way, the compiler statically resolves which function is to be called for any particular type.

3.5 Examples of Interpreters

Any instance of `Kanren` is called an interpreter, whether it evaluates a relation, transforms it, or does something completely different. In this subsection we will consider two examples of such interpreters.

For instance, consider the standard miniKanren interpreter over a stream of substitutions. The result of its execution, or representation, is `SubstKanren`. We add instances of `Kanren` and `KanrenEval` for the representation and provide the evaluation function `runSubstKanren`.

```
-- Interface of a substitution
emptySubst :: Subst s
readSubst :: V s a -> Subst s -> Maybe a
updateSubst :: V s a -> Maybe a -> Subst s -> Subst s

nextVar :: Subst s -> Int
succVar :: Subst s -> Subst s

-- Representation
newtype SubstKanren s a = SubstKanren (StateT (Subst s) Stream a)
                          deriving (Functor, Applicative, Alternative, Monad)

makeVar :: Maybe (L a (SubstKanren s)) -> R s (Var' a (R s))
makeVar x = do
    v <- SVar <$> gets nextVar
    modify $ succVar . updateSubst v x
    return $ v

type V s t = KVar (SubstKanren s) t
instance Kanren (SubstKanren s) where
    newtype instance (KVar (SubstKanren s)) t = SVar Int deriving (Functor, Show)

    fresh_ FreshVar   = (makeVar Nothing >>=)
    fresh_ (ArgVar x) = (makeVar (Just x) >>=)
    unify = flatteningUnify unif
        where
            unif v y | occursCheck v y = empty
                     | otherwise = do
                        x <- gets $ readSubst v
                        case x of
                            Nothing -> modify $ updateSubst v (Just y)
                            Just x' -> unify x' y
    call_ (name, (SubstKanren r)) = SubstKanren $ mapStateT Immature r

instance KanrenEval (SubstKanren s) where
    derefVar v = do
        x <- gets $ readSubst v
        case x of
            Nothing -> SubstKanren $ lift $ Immature $ generate
            Just x' -> eval x'

-- Evaluation function:
runSubstKanren :: (forall s. SubstKanren s t) -> Stream t
runSubstKanren (SubstKanren r) = evalStateT r emptySubst
```

Note that the substitution is a partial function that relies on `unsafeCoerce`, yet it is safe, as long as `SVar` constructor is not used by the programmer. Two measures enforce this safety: a

marker-parameter `s` (inspired by the implementation of `ST`), and the type tracking as part of a variable type. They ensure that the substitution always produces a value, and `unsafeCoerce` is applied only between identical types.

Now let us consider a normalizing interpreter which does not evaluate a relation, instead transforming it. It might be encoded in the following way:

```
data NormalizedBaseT (rel :: Type -> Type) where

    Unify :: (LogicType a) =>
             Var' a (NormalizedKanrenT rel) -> L a (NormalizedKanrenT rel)
          -> NormalizedBaseT rel
    Call :: String -> NormalizedKanrenT rel () -> NormalizedBaseT rel

newtype NormalizedConjT rel = Conj { unConj :: [NormalizedBaseT rel] }
data NormalizedRaiseT rel a = Raise { goal :: NormalizedConjT rel, retval :: a } deriving (Functor)
newtype NormalizedDisjT rel a = Disj { unDisj :: [NormalizedRaiseT rel a] } deriving (Functor)

data NormalizedFreshT rel a where

    FreshDone :: NormalizedDisjT rel a -> NormalizedFreshT rel a
    Fresh :: (LogicType x) =>
             FreshType (NormalizedKanrenT rel) x
          -> (Var' x (NormalizedKanrenT rel) -> NormalizedFreshT rel a) -> NormalizedFreshT rel a

-- Representation
type NormalizedKanrenT rel = NormalizedFreshT rel

-- Instance
instance (Kanren rel) => Kanren (NormalizedKanrenT rel) where
    newtype instance (KVar (NormalizedKanrenT rel)) a = NV (KVar rel a)

    fresh_ = Fresh
    unify = flatteningUnify (\x t -> FreshDone $ disj [raise (conj [Unify x t]) ()])

    call_ (Relation s r) = FreshDone $ disj [raise (conj [Call s r]) ()]

-- Evaluation function
normalize :: (Kanren rel) => NormalizedKanrenT rel a -> rel a
normalize = ...
```

We omit the instances of `Applicative` and `Alternative` which contain the normalization logic, because they are not important to the overall structure.

3.6 Extensibility Example

The major advantage of the tagless-final encoding is that additional structure can be introduced and composed directly in user code without any changes to the framework itself. Consider normalization from the previous subsection that can transform any relation without changing its observable type.

```
add :: (Kanren rel) =>
       Logic Nat (KVar rel) -> Logic Nat (KVar rel) -> Logic Nat (KVar rel)
    -> (String, rel ())
add = ...

addNorm :: (Kanren rel) =>
           Logic Nat (KVar rel) -> Logic Nat (KVar rel) -> Logic Nat (KVar rel)
        -> (String, rel ())
addNorm x y z = normalize <$> add x y z
```

However, here normalization is totally transparent. Even though the structure of the relation is more constrained, it is not exposed and cannot be used in further transformations, because the only interface available is `Kanren`. We can combat this issue by defining a new typeclass,

`NormalizedKanren`, to expose the normalized shape and make it interoperable with the `Kanren` typeclass.

```
class (Kanren (UnderlyingRel rel), Applicative (NRaise rel), Alternative (NDisj rel)) =>
      NormalizedKanren (rel :: Type -> Type) where

    type UnderlyingRel rel :: Type -> Type

    data NBase rel :: Type
    data NConj rel :: Type
    data NRaise rel :: Type -> Type
    data NDisj rel :: Type -> Type

    unifyVarNorm :: (LogicType a) =>
                    Var' a (UnderlyingRel rel) -> L a (UnderlyingRel rel) -> NBase rel

    callNorm_ :: (String, rel) -> NBase rel

    freshNorm_ :: (LogicType a) =>
                  FreshType (UnderlyingRel rel) a
               -> (Var' a (UnderlyingRel rel) -> rel x) -> rel x

    liftBase :: NBase rel -> NConj rel
    liftConj :: NConj rel -> a -> NRaise rel a
    liftRaise :: NRaise rel a -> NDisj rel a
    liftDisj :: NDisj rel a -> rel a

    makeConj :: [NBase rel] -> NConj rel
    makeDisj :: [NRaise rel a] -> NDisj rel a

interpretNorm :: (NormalizedKanren nrel) => NormalizedKanrenT (UnderlyingRel nrel) a -> nrel a
interpretNorm = ...
```

With this setup, any relation can be processed by a normalizing interpreter before being passed on to a conversion which will take full advantage of the normalized structure. Conversely, with an instance of `NormalizedKanrenT`, any normalized relation can be restored to the `Kanren` typeclass with the `normalize` function, erasing the structure. As a result, we demonstrated extensibility by providing two interoperable relation representations with no need to modify `Kanren`, or any other core feature.

4 Type-Preserving Functional Conversion

In this section, we present the updated functional-conversion pipeline that takes advantage of the typed miniKanren embedding. Rather than rebuilding the converter from scratch, we implemented an interpreter that lifts a shallowly embedded, typed relation into the deep, untyped representation used in the earlier work [9]. This choice lets us reuse the existing mechanisms of normalization, mode and determinism analyses, while still preserving limited type safety.

Even though the internal representation remains untyped, the fact that the translated relation originates from a typed embedded means that its underlying Haskell types as well as generators are available. This eliminates some inelegant aspects of the conversion process, namely synthesizing a monolithic *Term* type of all constructors, and threading explicit generators. Because of our assumption that the *quote* function (see 3.1) uses the actual constructor names of the underlying type, the converter can invoke them directly and call the type's *generate* whenever enumeration is required.

As an illustration, consider the relation $R_{\text{balance}}(x, y)$ that holds when the binary tree y is the balanced version of x.

```
balanceo :: (Kanren rel, LogicType elem) => L (Tree elem) rel -> L (Tree elem) rel -> Relation rel
```

```
balanceo = relation2 "balanceo" $ \v u -> fresh $ \e -> do
    call $ traverso v e
    call $ traverso u e
    call $ balancedo u
    pure ()

traverso :: (Kanren rel, LogicType elem) => L (Tree elem) rel -> L [elem] rel -> Relation rel
traverso = relation2 "traverso" $ \t e -> conde
    [ do
        t <=> leaf
        e <=> nil
        pure ()
    , fresh5 $ \l x r el er -> do
        t <=> node l x r
        call $ traverso l el
        call $ traverso r er
        call $ appendo el (cons x er) e
        pure ()
    ]

appendo :: (Kanren rel, LogicType elem) => L [elem] rel -> L [elem] rel -> L [elem] rel -> Relation rel
appendo = relation3 "appendo" $ \x y xy -> conde
    [ do
        x <=> nil
        y <=> xy
        pure ()
    , fresh3 $ \h x' xy' -> do
        x <=> cons h x'
        xy <=> cons h xy'
        call $ appendo x' y xy'
        pure ()
    ]

balancedo :: (Kanren rel, LogicType elem) => L (Tree elem) rel -> Relation rel
balancedo = relation "balancedo" $ \t -> conde
    [ t <=> leaf
    , fresh5 $ \l x r dl dr -> do
        t <=> node l x r
        call $ deptho l dl
        call $ deptho r dr
        call $ similaro dl dr
        call $ balancedo l
        call $ balancedo r
        pure ()
    ]

similaro :: (Kanren rel) => L Nat rel -> L Nat rel -> Relation rel
similaro = relation2 "similaro" $ \x y -> conde [ x <=> y, x <=> suc y, y <=> suc x ]

deptho :: (Kanren rel, LogicType elem) => L (Tree elem) rel -> L Nat rel -> Relation rel
deptho = relation2 "deptho" $ \t d -> conde
    [ do
        t <=> leaf
        d <=> zro
        pure ()
    , fresh3 $ \l x r -> fresh3 $ \ld rd d' -> do
        t <=> node l x r
        d <=> suc d'
        call $ deptho l ld
        call $ deptho r rd
        call $ maxo ld rd d'
        pure ()
    ]
```

```
65 ||
66 || lto :: (Kanren rel) => L Nat rel -> L Nat rel -> Relation rel
67 || lto = relation2 "lto" $ \l g -> conde
68 ||     [ fresh $ \g' -> do
69 ||         l <=> zro
70 ||         g <=> suc g'
71 ||         pure ()
72 ||     , fresh2 $ \l' g' -> do
73 ||         l <=> suc l'
74 ||         g <=> suc g'
75 ||         call $ lto l' g'
76 ||         pure ()
77 ||     ]
78 ||
79 || maxo :: (Kanren rel) => L Nat rel -> L Nat rel -> L Nat rel -> Relation rel
80 || maxo = relation3 "minmaxo" $ \x y mx -> conde
81 ||     [ mx <=> y *> call $ x `lto` (suc y)
82 ||     , mx <=> x *> call $ y `lto` x
83 ||     ]
```

We translate this relation in mode $R_{\text{balance}}\left(x^{\mathbf{Out}}, y^{\mathbf{Out}}\right)$, thus enumerating all trees alongside their balanced versions. The earlier functional conversion produces the following result. In addition to producing the monolithic *Term* type, it demonstrates the generator threading: *balancedoO_x2* is not used in *balanceoOO*, instead it is passed to *balancedoO*. Finally, it misses an optimization opportunity in line 14, where the *Stream* monad is used instead of the semi-deterministic *Maybe*.

```
1  || -- Monolithic term type
2  || data Term
3  ||     = Cons Term Term
4  ||     | Leaf
5  ||     | Nil
6  ||     | Node Term Term Term
7  ||     | S Term
8  ||     | Z
9  ||     deriving (Show, Eq)
10 ||
11 || -- Explicit generator threading
12 || balanceoOO balancedoO_x2 depthoOI_x3 = msum [do {x1 <- delay (balancedoO balancedoO_x2 depthoOI_x3);
13 ||                         -- Deterministic operation, but the computation is forced to use Stream
14 ||                                                 x2 <- delay (traversoIO x1);
15 ||                                                 x0 <- delay (traversoOI x2);
16 ||                                                 return (x0, x1)}]
17 || balancedoO balancedoO_x2 depthoOI_x3 = msum [do {x0 <- return Leaf;
18 ||                                                 return x0},
19 ||                                               do {x1 <- delay (balancedoO balancedoO_x2 depthoOI_x3);
20 ||                                                 x4 <- delay (depthoIO x1);
21 ||                                                 x5 <- delay (similaroIO x4);
22 ||                                                 x3 <- delay (depthoOI x5 depthoOI_x3);
23 ||                                                 delay (balancedoI x3);
24 ||                         -- Generator for a particular variable
25 ||                                                 x2 <- balancedoO_x2;
26 ||                                                 x0 <- return (Node x1 x2 x3);
27 ||                                                 return x0}]
28 ||
29 ||
30 || ...
```

By contrast, the type-preserving translation avoids creating the *Term* type and relies on the underlying types' constructors and the functions *generate*. An automatic determinism analysis allows us to use the *Maybe* monad whenever we can detect semi-deterministic computations. These computations are executed faster in *Maybe* and then lifted back to the *Stream* monad by *liftMaybe*.

```haskell
-- Constructors separated into underlying types
data Nat = Z | S Nat deriving (Eq, Ord, Show)
data Tree elem = Leaf | Node (Tree elem) elem (Tree elem) deriving (Eq, Ord, Show)

liftMaybe :: (Alternative m) => Maybe a -> m a
liftMaybe Nothing  = empty
liftMaybe (Just a) = pure a

-- No extra arguments
balanceo0O = msum [ do x1 <- delay balancedo0
            -- Computation converted to Maybe on a deterministic relation
                       x2 <- liftMaybe (traversoI0 x1)
                       x0 <- delay (traversoOI x2)
                       return (x0, x1)]
balancedo0 = msum [ do x0 <- return Leaf
                       return x0,
                    do x1 <- delay balancedo0
                       x4 <- delay (depthoI0 x1)
                       x5 <- delay (similaroI0 x4)
                       x3 <- delay (depthoOI x5)
                       delay (balancedoI x3)
            -- Generator is run from the inferred type
                       x2 <- generate
                       x0 <- return (Node x1 x2 x3)
                       return x0]
```

5 Experiments

To gauge the impact of our changes, we tested our functional conversion (New) against the untyped one (Old) on several relations. The new pipeline matches the old in every case and improves upon it in presence of semi-deterministic relations. We consider three relations: addition, generating permutations as a reverse of sorting, and typechecking used to enumerate terms of a given type. These relations showcase different aspects of performance gains.

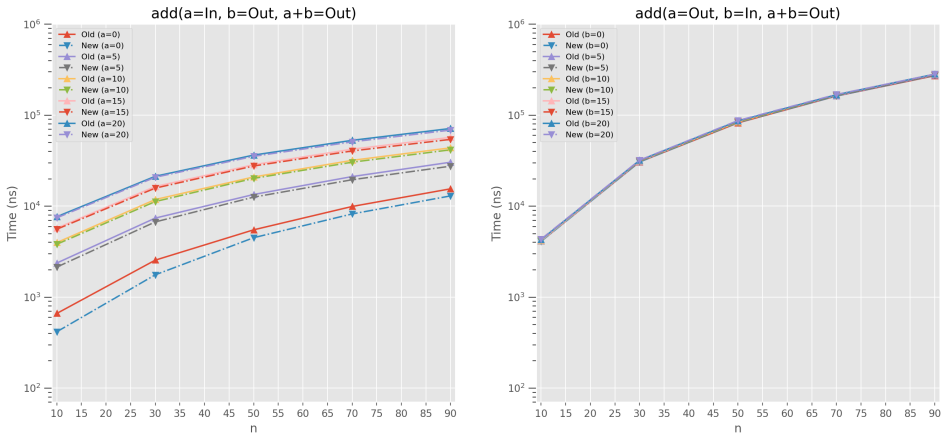

Fig. 4. Execution time (in nanoseconds) of *add* relation in two nondeterministic directions, requesting first n (x-axis) results. Log plot.

Figure 4 presents nondeterministic directions of addition, when exactly one variable is known. In the direction R_{add} (**In**, **Out**, **Out**) the relation uses one generator, and shows slight speed-up. The R_{add} (**Out**, **In**, **Out**) direction does not feature any optimization opportunities for our improved

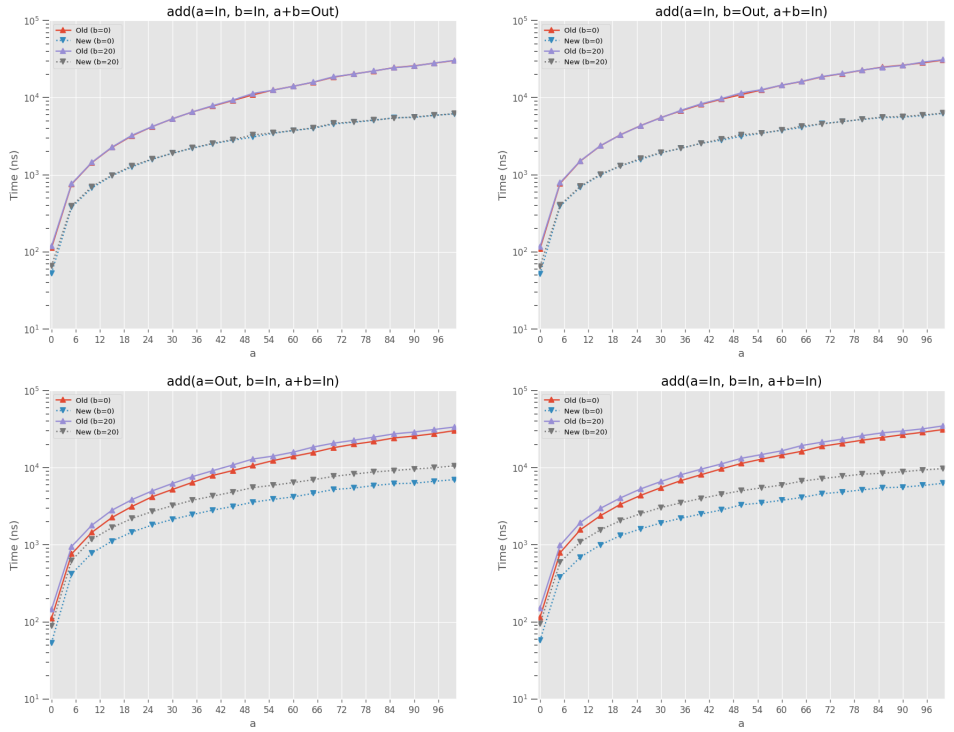

Fig. 5. Execution time (in nanoseconds) of *add* relation in four deterministic directions, with parameter *a* on x-axis. Log plot.

converter since it is nondeterministic, uses a single type, and avoids using generators. As a result, its performance matches the old implementation exactly.

Figure 5 covers semi-deterministic directions of addition. A clear performance improvement can be observed across all functions, reaffirming the importance of tracking determinism.

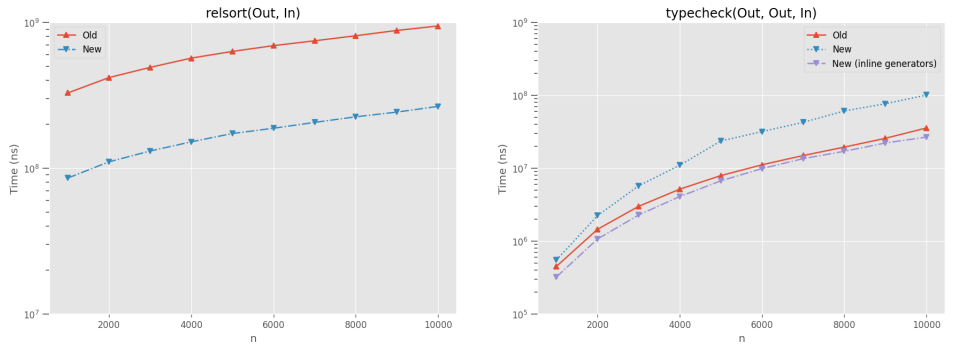

Fig. 6. Execution time (in nanoseconds) of generating permutations from 12-element list with the *sort* relation and enumerating terms and contexts that infer to a given type by the *typecheck* relation, requesting first *n* (x-axis) results. Log plot.

The left panel of Figure 6 showcases generating permutations of a sorted 12-element list by the *sort* relation. We observe performance improvement which arises entirely from semi-determinism of comparison relations.

The right panel of Figure 6 utilizes the *typecheck* relation to enumerate terms of a simple expression language that type check to the *Integer* type. This example demonstrates the importance of using `INLINEABLE` annotation for generators. When they are correctly inlined, we observe a reduced execution time, while naive implementation only worsens performance.

Overall, the new converter never slows down the relations. Moreover, in some cases, determinism analysis yields qualitative improvements by identifying computations which can safely be pruned once a single answer has been produced.

6 Conclusion

We presented a typed tagless-final embedding of miniKanren in Haskell and a functional conversion based on it. It addresses several inefficiencies and inelegant design decisions of the previous implementation, including monolithic *Term* types, generation threading and explicit determinism annotations. Moreover, the embedding promotes extensibility, one of the important aspects of the miniKanren language family.

References

[1] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 5 (2009), 509–543.

[2] Jason Hemann Daniel P Friedman. 2013. μKanren: A Minimal Functional Core for Relational Programming. In *Proceedings of the 2013 Workshop on Scheme and Functional Programming. http://webyrd.net/scheme-2013/papers/HemannMuKanren2013.pdf*. 4–5.

[3] Nikolai Kudasov and Artem Starikov. 2024. typedKanren: Statically Typed Relational Programming with Exhaustive Matching in Haskell. arXiv:2408.03170 [cs.PL] https://arxiv.org/abs/2408.03170

[4] Petr Lozov, Ekaterina Verbitskaia, and Dmitry Boulytchev. 2019. Relational interpreters for search problems. In *Relational Programming Workshop*. 43.

[5] Petr Lozov, Andrei Vyatkin, and Dmitry Boulytchev. 2018. Typed relational conversion. In *Trends in Functional Programming: 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers 18*. Springer, 39–58.

[6] 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*. Springer, 167–185.

[7] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. 1996. The execution algorithm of mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming* 29, 1 (1996), 17–64. doi:10.1016/S0743-1066(96)00068-4 High-Performance Implementations of Logic Programming Systems.

[8] Ekaterina Verbitskaia, Daniil Berezun, and Dmitry Boulytchev. 2021. An Empirical Study of Partial Deduction for miniKanren. In *Proceedings of the 9th International Workshop on Verification and Program Transformation, VPT@ETAPS 2021, Luxembourg, Luxembourg, 27th and 28th of March 2021 (EPTCS, Vol. 341)*, Alexei Lisitsa and Andrei P. Nemytykh (Eds.). 73–94. doi:10.4204/EPTCS.341.5

[9] Ekaterina Verbitskaia, Igor Engel, and Daniil Berezun. 2024. A case study in functional conversion and mode inference in minikanren. In *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation*. 107–118.

Received October 15, 2025