

# TYPEDKANREN: Statically Typed Relational Programming with Exhaustive Matching in Haskell

NIKOLAI KUDASOV, Innopolis University, Russia

ARTEM STARIKOV, Innopolis University, Russia

We present a statically typed embedding of relational programming (specifically a dialect of `MINIKANREN` with disequality constraints) in Haskell. Apart from handling types, our dialect extends standard relational combinator repertoire with a variation of relational matching that supports static exhaustiveness checks. To hide the boilerplate definitions and support comfortable logic programming with user-defined data types we use generic programming via `GHC.Generics` as well as metaprogramming via `Template Haskell`. We demonstrate our dialect on several examples and compare its performance against some other known implementations of `MINIKANREN`.

CCS Concepts: • **Theory of computation** → **Constraint and logic programming**; • **Software and its engineering** → **Functional languages**.

Additional Key Words and Phrases: relational programming, logic programming, functional programming, first-class modifiers, `miniKanren`, Haskell, generic programming, `Template Haskell`, pattern matching

## 1 INTRODUCTION

Haskell is a statically typed non-strict purely functional programming language that offers immense flexibility when it comes to support of programming paradigms. Indeed, while maintaining a functional core, its `do`-notation, which is overloaded via the `Monad` type class, provides an excellent environment for different kinds of imperative<sup>1</sup> programming. Additionally, Haskell offers a variety of generic programming tools [8, 25] which are often used to automate the boilerplate for the user when implementing domain-specific languages. The `do`-notation (as well as monad comprehensions [28]) can also be overloaded to offer logic programming capabilities (such as in `logict` [11]). However, unlike logic programming languages, Haskell does not provide unification variables or any support for unification of values out of the box.

Relational programming [21] is a kind of logic programming that relies on building programs as multiway relations. This approach allows using such relations as functions in different directions, depending on which of the arguments are known. A notable example is program synthesis from a relational interpreter [4].

`MINIKANREN`, introduced<sup>2</sup> by Byrd in his PhD thesis [3], is a family<sup>3</sup> of relational programming languages embedded into other languages. Most `MINIKANREN` implementations are untyped or untyped, meaning that all terms have the same type, regardless of what the terms represent. This follows the uniform representation approach popular in LISP-like languages as well as Prolog.

Almost all<sup>4</sup> `MINIKANREN` embeddings in Haskell that we know of also do not make use of the expressive type system that Haskell has to offer. Thus, we are aiming to fill in the gap and offer a complete statically typed `MINIKANREN` embedding in Haskell.

<sup>1</sup>Here, by “imperative” we mean programs that consist of sequences of instructions. These may, but need not support mutable state, exceptions, nondeterminism, and so on.

<sup>2</sup>Although core ideas of `MINIKANREN` can be traced to earlier work [11, 21].

<sup>3</sup>See a list of `MINIKANREN` implementations at <http://minikanren.org/#implementations>.

<sup>4</sup>The only exception we know of is `Molog` by Adam C. Foltzer which is an unfinished project with a similar approach to the one presented in this paper.

## 1.1 Related Work

Standalone typed logic programming languages such as Mercury and Curry exist. Remarkably, both languages exhibit an ML-style type system, very reminiscent of Haskell. A notable feature of Mercury’s type system is the *uniqueness types* that are used, in particular, to enable effects such as input/output. `TYPEDKANREN`, being embedded in Haskell, inherits many useful type system features. Although we do not yet explicitly allow effects in `TYPEDKANREN`, we believe our `Goal` monad can be easily extended into a monad transformer, similarly to `LogicT` [11].

There are typed embeddings of `MINIKANREN` into OCaml (`OCANREN` [13]), Kotlin (`KLOGIC` [9]), and Rust (`canrun_rs` [26]). To the best of our knowledge, none of these implementations provide a typed version of `match`<sup>e</sup>. `TYPEDKANREN` provides a typed version of regular matching as well a version with *static* exhaustiveness checks.

## 1.2 Contribution

In this paper, we present an embedding of `MINIKANREN` into Haskell. Our dialect, named `TYPEDKANREN`, enables typed relational programming in Haskell with typed unification. Specifically, our contribution is as follows:

- (1) In Section 2, we design the relational engine at the core of `TYPEDKANREN`. For the underlying representation we follow the classical `MINIKANREN` implementations like `μKANREN` [7], however, we also set up a system of type classes that enables typed relational programming on the user side.
- (2) In Section 3, we implement matching, discuss differences between functional and relational matching. Here we also introduce a new combinator for *exhaustive* relational matching, enabling better safety and performance for some relational programs.
- (3) In Section 4, we use generic programming and metaprogramming techniques to hide most of the boilerplate away from the user, so that user-defined data types could be easily used in a relational program.
- (4) In Section 5, we compare performance of our implementation against other `MINIKANREN` implementations, including `faster-minikanren` [2], `OCANREN` [13], and `KLOGIC` [9].

The implementation of `TYPEDKANREN` is available on GitHub at [github.com/SnejUgal/typedKanren](https://github.com/SnejUgal/typedKanren). A separate repository with combined benchmarks for `faster-minikanren`, `OCANREN`, `KLOGIC`, and `TYPEDKANREN` is available at [github.com/SnejUgal/typedKanren-benchmarks](https://github.com/SnejUgal/typedKanren-benchmarks).

## 2 CORE

In this section, we introduce the core of `TYPEDKANREN` and describe some design and implementation choices.

From the user perspective, `TYPEDKANREN` consists of

- (1) The `Logical` typeclass that specifies the types that may enter the relational world; specifically, these are the types that have their logical counterparts, support unification, and conversion between data representations for functional and relational programs.
- (2) The `Goal` monad, providing the context for the relational programs, together with the goal combinators, including unification, disunification (disequality), conditional and matching operators, and fresh variable generators.
- (3) Automation utilities that help making most user-defined types `Logical`.

Ignoring the types, our implementation follows closely the classical implementations of `MINIKANREN` [4, 7]. That said, our implementation is in Haskell, a non-strict (or “lazy”) functional language, which means that we do not have to rely on the explicitly constructed thunks, making the core implementation somewhat more direct, in our opinion.

## 2.1 Logical Types and Unification Terms

The objects of relational programming in TYPEDKANREN are *logical* types. Values of such types may admit unification variables in places where a regular Haskell type would always have a regular specified value. For example, consider the following recursive parametrically polymorphic type of trees:

```
data Tree a
= Empty           -- ^ An empty tree.
| Leaf a          -- ^ A leaf holding a value.
| Node (Tree a) (Tree a) -- ^ A node with two subtrees.
```

Here, the **Leaf** constructor has an associated value (a field) of type **a** and **Node** has two fields of type **Tree a**. When going into the relational world, we might expect the tree to be underspecified, having an undetermined unification variable in place of one or both of the branches. Therefore, relational programming demands a similar, but separate data type, a relational counterpart to **Tree**:

```
data LogicTree a
= LogicEmpty
| LogicLeaf (Term a)
| LogicNode (Term (Tree a)) (Term (Tree a))
```

The type constructor **Term** provides the logical counterpart to its argument, except the whole term is also allowed to be a unification variable:

```
data Term a
= Var !(VarId a) -- ^ A unification variable.
| Value !(Logic a) -- ^ A logical version of type a.
```

The variable identifiers **VarId** are parametrized with a phantom type parameter to avoid accidental confusion between unification variables for terms of different types. Under the hood, a variable identifier is merely a machine-sized integer:

```
newtype VarId a = VarId Int
```

The type family **Logic** maps types to their relational counterparts. For example, **Logic (Tree a)** is the same as **LogicTree a**. To keep track and allow extending this correspondence, we introduce the **Logical** typeclass:

```
class Logical a where
  type Logic a = r | r -> a
  unify :: Logic a -> Logic a -> State -> Maybe State
  walk  :: State -> Logic a -> Logic a
  occursCheck :: VarId b -> Logic a -> State -> Bool
  inject :: a -> Logic a
  extract :: Logic a -> Maybe a
```

Although the typeclass formally requires a type family and five methods, our main focus lies only with the type family, since the methods normally have standard implementation which is derived automatically via **GHC.Generics** [8], an automation process described in more detail in Section 4.2.

Technically, the logical counterparts to user-defined types may also be generated automatically with Template Haskell [25], as is described in Section 4.1. That said, while the methods of **Logical** are normally used in the background, the logical types face the user, in particular when dealing with type errors.

Note that in the definition of type family **Logic** we impose a functional dependency, which ensures that the original type **a** can always be uniquely recovered by the compiler if it knows the logical type. This is required, since most of the methods deal only with the **Logic** types and without the functional dependency the instance of **Logical** may become ambiguous.

Now, as we have established above, **Tree** *a* has its logical counterpart, so we can implement<sup>5</sup> an instance of **Logical**:

```
instance Logical a => Logical (Tree a) where
  type Logic (Tree a) = LogicTree a
  ...
```

It is important that the type parameter *a* is also **Logical**, since the definition of **LogicTree** makes use of **Term** *a*, which in turn relies on **Logic** *a*, which is only defined when *a* is **Logical**.

## 2.2 Unification of Terms

The main job of the relational engine of **TYPEDKANREN** is to keep track of what is known about the terms in a relational program (possibly in different parallel branches). This information is tracked in the state of the engine and consists of the substitution for unification variables and the disequality constraints.

Following classical **MINIKANREN** implementations [4, 7], we implement “triangular” substitutions [1, §2.2.6] with an “occurs” check in the unification procedure. We rely on integers to represent unification variables, and use efficient purely functional integer maps [19] to represent substitutions.

In our implementation, we keep unification variables for all types in a single map. To achieve that, we use existential types to temporarily “forget” the type of a term<sup>6</sup>:

```
data ErasedTerm where
  ErasedTerm :: Logical a => Term a -> ErasedTerm
```

After forgetting the types, we can keep all substitutions in one place:

```
newtype Subst = Subst (IntMap ErasedTerm)
```

Of course, when looking up a variable in a substitution, we no longer know the type of the term. However, since the variable identifiers facing the user are always annotated with the proper type (and the user cannot safely coerce), we are able to hide the only<sup>7</sup> unsafe coercion and wrap it into a safe function:

```
lookupSubst :: VarId a -> Subst -> Maybe (Term a)
lookupSubst = unsafeCoerce IntMap.lookup
```

The remaining implementation of the unification is a straightforward adaptation of the standard approach [4, §D.3], so we omit it here, referring the reader to the source code of modules **Kanren.Core** and **Kanren.Goal**.

The disequality constraints follow **MINIKANREN** [4, §D.1]. Each constraint is bound to a single unification variable, and the overall set of constraints is represented with the following data type:

```
newtype Disequalities = Disequalities (IntMap [(ErasedTerm, Subst)])
```

Again, we provide a safe extraction that uses unsafe coercion under the hood. Otherwise, the implementation of disequality constraints is standard.

## 2.3 The Goal Monad

Relational programs in **TYPEDKANREN** consist largely of programming with *goals*, essentially in the same way as in other **MINIKANREN** implementations. However, since Haskell provides great syntactical support for imperative-looking code via the **do**-notation, it is very convenient to upgrade goals to a proper monadic context.

<sup>5</sup>We omit the implementation of methods here, see Section 4.2 for the details of generic implementations for them.

<sup>6</sup>In this definition, we preserve the knowledge that *a* is **Logical**. While this is not critical for the unification, it plays a role in the implementation of disequality constraints.

<sup>7</sup>A similar lookup is needed in the implementation of disequality constraints, but the two can be refactored to only require a single call to **unsafeCoerce**, retaining all other properties.

We define the **Goal**  $x$  to be the function that when given an initial state generates a stream of possible states (satisfying the relations of the goal) each with a value of type  $x$  associated with it:

```
newtype Goal x = Goal {runGoal :: State -> Stream (State, x)}
```

We define the “immature streams” [7, §4.2] here in a standard way, except relying on an algebraic data type with lazy constructors instead of explicit thunks:

```
data Stream a
= Done
| Yield a (Stream a) -- "mature" stream
| Await (Stream a)   -- "immature" stream
```

Naturally, **Stream** and **Goal** possess the structure of a monad. We implement the corresponding **Monad** instances, following the standard definition [4, §D.1], relying on the interleaving of streams.

The **State** simply contains information about substitutions, disequality constraints, and a global counter used to generate fresh unification variables:

```
data State = State
{ knownSubst    :: !Subst
, disequalities :: !Disequalities
, maxVarId      :: !Int
}
```

We provide the following basic goal constructors:

- (1) **successo** ::  $x \rightarrow \text{Goal } x$  is a goal that always succeeds. This is the same as **return** (from the **Monad** typeclass).
- (2) **failo** :: **Goal**  $x$  is a goal that always fails.
- (3) **(==)** :: **Logical**  $a \Rightarrow \text{Term } a \rightarrow \text{Term } a \rightarrow \text{Goal } ()$  is a goal that unifies the two given terms of type  $a$ .
- (4) **conj** :: **Goal**  $x \rightarrow \text{Goal } y \rightarrow \text{Goal } y$  represents the conjunction of two goals. The result of the first goal is ignored<sup>8</sup>, while the result of the second goal is kept. This corresponds to **(>>)** (from the **Monad** typeclass), and more general versions of conjunction include **ap** and **>>=**. A version of **conj** that conjuncts a list of goals is **conjMany**.
- (5) **disj** :: **Goal**  $x \rightarrow \text{Goal } x \rightarrow \text{Goal } x$  represents the disjunction of two goals. The streams from the two goals are interleaved. A version of **disj** that disjuncts a list of goals is **disjMany**.
- (6) **conde** ::  $[[\text{Goal } ()]] \rightarrow \text{Goal } ()$  is TYPEDKANREN’s version of **cond**<sup>e</sup>: the outer list represents alternatives (disjunctions), and inner lists represent conjunctions.
- (7) **(=/=)** :: **Logical**  $a \Rightarrow \text{Term } a \rightarrow \text{Term } a \rightarrow \text{Goal } ()$  is a goal that disunifies the two given terms of type  $a$ .
- (8) **fresh** :: **Fresh**  $v \Rightarrow \text{Goal } v$  is a goal that produces a fresh unification variable of a given type.

Regarding **fresh**, the user is not expected to implement any new instances of **Fresh** typeclass. This is because there exists an instance for logical terms as well as tuples of logical terms:

```
instance (Logical a) => Fresh (Term a) where ...
instance (Logical a, Logical b) => Fresh (Term a, Term b) where ...
```

Finally, to run the relational program, we execute the goal:

```
run :: Fresh v => (v -> Goal ()) -> [v]
```

This function produces a *lazy* list of values (usually of type **Term**  $a$  for some  $a$ ).

<sup>8</sup>Hence, the type of the first goal can be different from the second goal and the result type.

## 2.4 Example

With all the basic ingredients in place, we can implement a simple relational program. First, consider a functional program that extracts all leaf values from a **Tree**:

```
leaves :: Tree a -> [a]
leaves t = case t of
  Empty -> []
  Leaf x -> [x]
  Node l r -> leaves l ++ leaves r
```

Converting this into a relational program we get

```
leaveso :: Logical a => Term (Tree a) -> Term [a] -> Goal ()
leaveso t xs = disjMany
  [ do
    t === Value LogicEmpty
    xs === Value LogicNil
  , do
    x <- fresh
    t === Value (LogicLeaf x)
    xs === Value (LogicCons x (Value LogicNil))
  , do
    (l, r, as, bs) <- fresh
    t === Value (LogicNode l r)
    leaveso l as
    leaveso r bs
    appendo as bs xs
  ]
```

In this example, we use `disjMany` with a list of `do`-blocks, as we see it less syntactically noisy compared to `conde` in `TYPEDKANREN`. The constructors `LogicNil` and `LogicCons` are the logical counterparts to the empty and non-empty list constructors<sup>9</sup>.

It is possible to run this relational program in both directions. First, we may specify the tree and ask `TYPEDKANREN` to compute all possible lists corresponding to the leaves of a given tree (we expect exactly one possibly):

```
>>> t = Node (Node (Leaf 1) Empty) (Leaf (2 :: Int))
>>> run (leaveso (inject' t))
[Value (LogicCons (Value 1) (Value (LogicCons (Value 2) (Value LogicNil))))]
```

We may extract the non-logical value(s) (safely, using `Maybe`):

```
>>> extract' <$> run (leaveso (inject' t))
[Just [1,2]]
```

Specifying the list and leaving the tree as an argument, we effectively run the relation “backwards”. Here, we take and print the first five<sup>10</sup> trees that have two leaves with values 1 and 2 (in that order):

```
>>> mapM_ (print . extract') $ take 5 (run (`leaveso` (inject' [1, 2 :: Int])))
Just (Node (Leaf 1) (Leaf 2))
Just (Node Empty (Node (Leaf 1) (Leaf 2)))
Just (Node (Leaf 1) (Node Empty (Leaf 2)))
Just (Node (Leaf 1) (Node (Leaf 2) Empty))
Just (Node (Node Empty (Leaf 1)) (Leaf 2))
```

<sup>9</sup>It is possible to rely on `OverloadedLists` extension to use regular syntax for lists, but we prefer to use explicit constructors here for clarity of presentation.

<sup>10</sup>Since some leaves may be `Empty`, there are infinitely many trees corresponding to the list `[1, 2]`.

### 3 EXHAUSTIVE MATCHING

In his thesis [3], Byrd introduces a logical matching operator **match**<sup>e</sup>, a relational counterpart to the functional matching operators **pmatch** [3, Appendix B] and **dmatch** [4, Appendix C]. Of course, TYPEDKANREN also provides **matche**, however, in this section, we explore the design space for statically typed relational matching operators, considering both safety and efficiency.

#### 3.1 Prisms as First-Class Patterns in Haskell

Unlike LISP, quasiquotation is not built into Haskell<sup>11</sup>, and thus patterns and data constructors only work in one direction at a time: when used in a **case**-expression or in function argument the data constructor can be used to match against a Haskell value, while when used in an expression, the same data constructor is used to create a Haskell value.

In a relational setting, we want to work with patterns that work simultaneously in both directions, relying on unification under the hood. To achieve this, we use the well-known technique for first-class patterns in Haskell — the *prisms*. Semantically, a simple prism of type **Prism** *s a* is a Haskell value that is equivalent to a pair of functions for matching (of type *s -> Maybe a*) and constructing values (of type *a -> s*). The two common representations for prisms are the van Laarhoven representation [27], used in the Kmett’s **lens** library [12], and the profunctor optics representation [22]. The former is widely used in Haskell, so we go with it in this paper. However, we do not see any reason that profunctor optics would not work just as well in this setting.

An important generalization for prisms (as well as other optics), is a separation of types for matching and construction. Specifically, a prism of type **Prism** *s t a b* semantically consists of a matching function of type *s -> Either t a* and a construction function of type *b -> t*. This separation of types is important as it allows changing types when modifying a value under a prism or enforce further properties on the prisms.

Below we present two examples, relevant to the design of typed relational matching in TYPED-KANREN. Both examples will rely on the following definition and relevant prisms<sup>12</sup>:

```
data Result a b = Ok a | Fail b
```

```
_Ok   :: Prism (Result a c) (Result b c) a b
```

```
_Fail :: Prism (Result c a) (Result c b) a b
```

**3.1.1 Changing Types with Prisms.** In the following example, we use the prism **\_Fail** to modify the type of errors from **Err** to **String** by using a standard Haskell function **show** to convert a value into a string:

```
example :: Result Int Err -> Result Int String
```

```
example result = result
```

```
  & _Ok   %~ (+1) -- increment result if it is Ok
```

```
  & _Fail %~ show -- convert error into a String if it is Fail
```

**3.1.2 Eliminating Alternatives with Prisms.** In this next example, we use the ability of the prism **Ok** to change the type of its corresponding constructor to any other type when it does not appear in the value we are matching against:

```
example :: Result Int Bool -> Int
```

```
example result =
```

<sup>11</sup>Quasiquotation is supported by Template Haskell, however, it is unclear how to properly use it for patterns here, since we want the main combinators to be regular Haskell functions, not Template Haskell functions (which would correspond to LISP macros). So, we leave research into feasibility of quasiquotation for future work.

<sup>12</sup>We omit the implementation of prisms, as only the types are important in the discussion and implementation may vary depending on the chosen representation.



```

case matching _Ok result :: Either Void Bool of
-- successful matching on Ok, extracting n :: Int
Right n -> n
-- matching failed, x :: Either Void Bool
Left x -> case x of
-- Fail is still a possible alternative
Fail b -> fromEnum b
-- Ok is impossible here, hence the use of absurd
Ok n -> absurd n

```

### 3.2 Typed Relational Matching

The prism-based relational matching is presented in `TYPEDKANREN` with two combinators.

First, `matche :: Term a -> Goal ()` is the default matching combinator without any pattern matching branches.

Second, `on` combinator acts as a modifier, adding a single pattern matching branch to an existing matching; the type of `on` is fairly straightforward:

```

on :: (Logical a, Fresh v)
=> Prism' (Logic a) v -- ^ The pattern presented as a prism.
-> (v -> Goal x)      -- ^ The handler for this one alternative.
-> (Term a -> Goal x) -- ^ Matching for other alternatives.
-> (Term a -> Goal x) -- ^ Extended matching (+1 alternative).

```

Technically, all branches here have the same type and could be organized in a list, resembling more the traditional `match`<sup>e</sup>. However, we choose the approach of modifiers for matchers, as it generalizes well for the safer exhaustive matching variant discussed below, following the design of Gonzalez's `total` library [5, 6].

As an example, consider a relational version of list concatenation in `TYPEDKANREN`:

```

-- the following prisms are used to match on lists in a relational program
_LogicNil :: Prism' (Logic [a]) ()
_LogicCons :: Prism' (Logic [a]) (a, Term [a])

appendo :: Logical a => Term [a] -> Term [a] -> Term [a] -> Goal ()
appendo xs ys zs = xs & (matche -- matching on xs
  & on _LogicNil (\() -> ys === zs) -- if xs is empty, then unify ys and zs
  & on _LogicCons (\(x, xs') -> do -- if xs is non-empty with head x and tail xs'
    zs' <- fresh -- then
    zs === Value (LogicCons x zs') -- zs is also nonempty with head x and tail zs'
    appendo xs' ys zs')) -- such that xs' ++ ys unifies with zs'

```

It might be useful to compare it with a functional version<sup>13</sup> (working in one direction only):

```

append :: [a] -> [a] -> [a]
append xs ys = case xs of
[] -> let zs = ys in zs
(x:xs') ->
  let zs = x : zs'
      zs' = append xs' ys
  in zs

```

We note that the structure of the program is similar:

- `matche` and `on` replace the `case`-expression in the relational version;

<sup>13</sup>We use `let`-bindings for some (sub)expressions to underline the similar computational structure to the relational version.



- `fresh` and binding in a `do`-block is used instead of `let`
- in the relational program, the order of goals in the `_LogicCons`-branch matters while the order of `let`-bindings in the functional program does not; while there are approaches to fair relational conjunction [11, 15, 17], TYPEDKANREN does not implement those variants.

Although we have achieved typed relational matching, which works well for many occasions, we believe, it presents a few disadvantages that can be eliminated, at least for some relational programs:

- (1) Under the hood, each matching branch contributes a separate runtime-matching, which is akin to a series of nested `if`-expressions. This is, of course, suboptimal: our `matche` has (under the hood) a `case`-expression per matching branch, whereas the functional version only has one overall `case`-expression for the entire match. To the best of our knowledge, other implementations of MINIKANREN, including the statically typed `OCANREN`, rely on unification and `cond`<sup>e</sup>, similarly resulting in multiple redundant matches on the input value.
- (2) Many relational programs, similarly to `appendo` above, still consider the full set of distinct (non-overlapping) patterns in the matching branches. At the same time, in Haskell, and many other statically typed languages, exhaustiveness checks for the pattern matching (e.g. in `case`-expressions) are considered a vital resource for making sure no important branch is forgotten, especially when refactoring the code. Therefore, it appears to be practically useful to allow users enable exhaustiveness checking for relational matching.

### 3.3 Exhaustive Relational Matching

In this section, we describe a variation of relational matching with exhaustiveness checks, enabling safer relational programs.

Since we use prisms as first-class patterns for the relational matching, we cannot rely on the built-in exhaustiveness checker in Haskell, since it operates only with Haskell's native patterns, not prisms. Gonzalez [5, 6] has proposed a mechanism for exhaustive matching that works with lenses, prisms, and traversals. In TYPEDKANREN, we are following along the same ideas, except it becomes slightly more involved since we are using prisms in both directions.

The main idea is to introduce a version of `on` that visibly reduces the possible alternatives for matching tracked in the types. A first approximation is as follows:

```
on1 :: (Logical a, Fresh v)
    => Prism (Logic a) (Logic b) v Void
    -> (v -> Goal x)
    -> (Term b -> Goal x)
    -> (Term a -> Goal x)
```

The generalized form of the prism ensures that the type `Logic b` is exactly a version of the type `Logic a` where an alternative containing `v` is impossible (replaced with `Void`). A specialized version of this is exactly what is used in Gonzalez's `total` library [5].

Unfortunately, using `Void` in the last type parameter of the prism effectively forbids using this prism for construction. Indeed, the construction function extracted from such a prism would have the type `Void -> Logic b`, which is impossible to apply while staying in the safe Haskell territory.

To preserve the ability to construct values and keep track of the remaining alternatives, in TYPEDKANREN we use the type-level annotations via the `Tagged` wrapper type:

```
-- | A value of type b, tagged with type a.
newtype Tagged a b = Tagged b
```

**3.3.1 Patterns Tagged with Exhaustiveness Information.** In the context of exhaustive relational matching, the type-level tags carry the information about checked and remaining cases. To keep track of this information, we use special version of the logical prisms for the tagged logical values.

For example, such annotated prism for the **Ok** constructor of **Result** has the following type and is defined as a coercion over the regular prism, imposing zero runtime cost<sup>14</sup>:

```
_LogicOk' :: Prism
  (Tagged (ok, fail) (Logic (Result a b)))
  (Tagged (ok', fail) (Logic (Result a b)))
  (Tagged ok (Term a))
  (Tagged ok' (Term a))
_LogicOk' = from _Tagged . _LogicOk . _Tagged
```

Let us dissect the type arguments of this prism:

- (1) **Tagged** (ok, fail) (**Logic** (**Result** a b)) corresponds to the matched logical value of type **Result** a b. The tags **ok** and **fail** designate the information about checked (sub)cases for the data constructors **Ok** and **Fail** respectively. For example, when (ok, fail) is (**Remaining**, **Checked**), that would mean that only the **Ok** case remains to be checked.
- (2) **Tagged** (ok', fail) (**Logic** (**Result** a' b)) corresponds to the logical value for the remaining cases (*after* considering the **Ok** case). The tag **ok'** corresponds to the updated information about the **Ok** case. If matching concerns the entire **Ok** constructor (regardless of the subcases for the type a), then we would expect **ok'** to be exactly **Checked**. However, the type signature of **\_LogicOk'** allows **ok'** to be partially checked, in case of nested prisms.
- (3) **Tagged** ok (**Term** a) corresponds to the type of the logical term<sup>15</sup> for the chosen case *before* handling this alternative.
- (4) **Tagged** ok' (**Term** a') corresponds to the type of the logical term for the chosen case *after* handling this alternative. Note that both the tag and the type are allowed to be changed (since **Result** is parametrically polymorphic in the contents of its **Ok** constructor).

For conciseness, we introduce the **ExhaustivePrism** type alias. The type of **\_LogicOk'** then becomes:

```
_LogicOk' :: ExhaustivePrism
  (Logic (Result a b)) (ok, fail) (ok, fail')
  (Term a) ok ok'
```

**3.3.2 Exhaustive Relational Matching.** We define three combinators for exhaustive matching.

First, we define **on'** which is similar to **on**, but makes use of annotated prisms. Its type is more complicated:

```
on'
  :: (Logical a, Fresh v)
  => ExhaustivePrism (Logic a) m m' v Remaining Checked
  -- ^ The pattern presented as an annotated prism.
  -> (v -> Goal x) -- ^ The handler for this one alternative.
  -> (Matched m' a -> Goal x) -- ^ Matching for other alternatives.
  -> (Matched m a -> Goal x) -- ^ Extended matching.
```

**on'** specializes the prism so that its case is marked as **Checked**. The changed list of tags is then passed to other alternatives. The **Remaining** tag demands this case not be checked previously;

<sup>14</sup>Since **Tagged** is a **newtype**, its runtime representation is identical to the underlying type and corresponding coercion functions are identities that we trust are optimized away by the Glasgow Haskell Compiler. For guaranteed zero cost abstraction, **unsafeCoerce** can be (safely) used here.

<sup>15</sup>Note that it also includes the case for just a unification variable, which is indicated by the use for **Term** instead of **Logic**.

although we could allow checking the same pattern twice, this would pose inconvenient edge cases in nested matching. `Matched m a` is an alias for `Tagged m (Term a)`.

Second, we define `matche'` which is similar to `matche`, but also performs exhaustiveness check by constraining `m` to contain only `Checked` tags (by imposing `Exhausted m` constraint):

```
matche' :: Exhausted m => Matched m a -> Goal x
```

Finally, we define `enter'` which is a new combinator attaching tags to the term being matched.

```
enter' :: (Matched m a -> Goal x) -> Term a -> Goal x
```

As an example, consider the following relational program. For clarity, we show tags (`m`) passed to each combinator.

```
resulto :: (Logical a, Logical b) => Term (Result a b) -> Goal ()
resulto r = r & (matche'          -- ( Checked,  Checked)
  & on' _LogicOk' (\_ -> successo ()) -- (Remaining, Checked)
  & on' _LogicFail' (\_ -> successo ()) -- (Remaining, Remaining)
  & enter')
```

Now consider what happens when an alternative is omitted. If so, its corresponding tag is never instantiated to a particular type and remains a type variable.

```
resulto :: (Logical a, Logical b) => Term (Result a b) -> Goal ()
resulto r = r & (matche'          -- (ok,    Checked)
  & on' _LogicFail' (\_ -> successo ()) -- (ok, Remaining)
  & enter')
```

The constraint `Exhausted m` imposed by `matche'` will reduce to `Exhausted ok`. However, since `ok` is a type variable not constrained by anything, this constraint will not be satisfied and the program will fail to compile.

## 4 HIDING THE BOILERPLATE

Here we explain how the boilerplate definitions, helper functions, and class instances can be generated with `GHC.Generics` [8] and `Template Haskell` [25].

### 4.1 Generating the Logical Types

For each Haskell data type, there may exist many logical variants, depending on where we allow unification variables to occur. By default, and in the examples in the previous sections, we have allowed unification variables to occur anywhere. More specifically, for every data constructor, each of the fields of type `T` in the original data type becomes a field of type `Term T` in the relational version of that type. This is a very straightforward approach, that maximizes the use of unification variables, and can be easily automated.

We use `Template Haskell` [25] to generate such *maximal* logical types together with the appropriate instances. For mutually recursive types, we also provide more customizable functions to separately generate types and instances. Specifically, we provide the following template functions:

```
makeLogic      :: Name -> Q [Dec]  -- type and instances
makeLogicType  :: Name -> Q [Dec]  -- only type
makeLogicTypes :: [Name] -> Q [Dec] -- mutually recursive types

-- generate a default Generic-based Logical instance(s)
makeLogicalInstance :: Name -> Name -> Q [Dec]
makeLogicalInstances :: [(Name, Name)] -> Q [Dec]
```

Given the name of a user-defined data type, we generate its maximal logical counterpart, as well as a generic **Logical** instance. For example, consider the following user code defining a **Tree** data type and invoking `makeLogic` for it:

```
{-# LANGUAGE TemplateHaskell #-}
...
data Tree a
  = Leaf a
  | Node (Tree a) (Tree a)
  deriving (Eq, Show, Generic)
makeLogic ''Tree
```

Our Template Haskell code systematically inspects the user-defined data type declaration and generates the following code:

```
data LogicTree a
  = LogicLeaf (Term a)
  | LogicNode (Term (Tree a)) (Term (Tree a))
  deriving (Generic)

instance Logical a => Logical (Tree a) where
  type Logic (Tree a) = LogicTree a
  unify = genericUnify
  subst = genericSubst
  occursCheck = genericOccursCheck
  inject = genericInject
  extract = genericExtract
```

The `generic*` functions are explained in detail in Section 4.2. Importantly, `makeLogic` supports the vast majority of valid Haskell data type definitions, including records, infix constructors, strictness annotations<sup>16</sup>, and generalized algebraic data types. It should be noted though, that generalized algebraic data types might not have `GHC.Generic` instances. Support for TH-generated implementations of **Logical** methods for such cases is not present and is subject to future work.

For our code to work, the following language extensions are required on the user side in addition to `TemplateHaskell: DeriveGeneric, TypeFamilies`.

Implementation of `makeLogic` makes the following implicit decisions that are not configurable at the moment, pending future work:

- (1) the type name and all (prefix) constructor names are prefixed with **Logic**: `Foo` becomes `LogicFoo`;
- (2) field names are prefixed with `logic` and capitalized: `fooBar` becomes `logicFooBar`;
- (3) all fields are wrapped in **Term**: `T` becomes `Term T`.

## 4.2 Generic Unification for Logical Types

To enable unification for the logical types, `TYPEDKANREN` requires the following methods of the **Logical** typeclass to be implemented:

- (1) `unify` performs unification of two logical values;
- (2) `walk` updates a logical value, applying known substitutions to any unification variables in the logical value;
- (3) `occursCheck` inspects a logical value to see if a given variable is present in one of its subterms;

<sup>16</sup>Since both constructors of `Term a` are defined as strict, preserving strictness annotations in logical versions of user-defined constructors does not alter the intended strictness of terms.

- (4) `inject` converts regular value of type `a` into a logical value of type `Logic a` (without any variables);
- (5) `extract` attempts to convert a logical value of type `Logic a` to the regular value of type `a`; of course, since it is not possible in presence of unification variables, the result is wrapped in a `Maybe`.

Fortunately, all of these methods follow one of the two standard implementations:

- (1) For base types, such as `Int` or `Bool` for which their logical counterpart coincides with the regular type, unification simplifies to a mere equality check, `walk` and `inject` are identities, `occursCheck` is always false (there are no unification variables allowed), and `extract` simply wraps input in a `Just` constructor.
- (2) For cases when logical type is distinct, their structure is expected to match: constructor-to-constructor and field-to-field. In this case, we can rely on the generic structure of algebraic types (as provided by `GHC.Generics`) to derive the corresponding methods. This approach is described below in detail.

We introduce a generic counterpart to `Logical` typeclass. The idea is to implement all necessary methods for a generic representation and then, for user-defined types, implement each method via conversion back and forth between the generic representation and the user code. Importantly, relying on the generic representation and such conversions is a zero-cost abstraction in GHC [8], providing a safer (compared to Template Haskell) and efficient generic implementation for the `Logical` methods.

The generic logical typeclass takes two parameters: the representation types of the user-defined type and its logical counterpart. We found the multiparameter definition to be somewhat easier to work with here, although it is technically possible to use an associated type family as in `Logical`. Note the use of `Proxy` `d` to ensure we always know which type `f` the method corresponds to. This is the alternative to the functional dependency used in the definition of the associated type family `Logical`.

```
class GLogical f f' where
  gunify :: Proxy f -> f' p -> f' p -> State -> Maybe State
  gwalk :: Proxy f -> State -> f' p -> f' p
  goccursCheck :: Proxy f -> VarId b -> f' p -> State -> Bool
  ginject :: f p -> f' p
  gextract :: f' p -> Maybe (f p)
```

For the generic implementation, it is sufficient to consider the following cases:

- (1) `V1` — representation for an empty type;
- (2) `U1` — representation for a unit type (think of a constructor without fields);
- (3) `f :*` `g` — representation for a product of types (think of a constructor with at least one field);
- (4) `f :+:` `g` — representation for a sum of types (think of a data type with at least one constructor);
- (5) `K1 i c` — representation for a field of type `c`;
- (6) `M1 i t f` — same as `f`, except with some metadata over it.

To illustrate the generic implementation, we demonstrate the implementation of `gunify` for the specified cases above. The empty and unit types are trivial, as there is nothing to check, and we can say that values of these types always unify without any changes to the state:

```
instance GLogical V1 V1 where gunify _ _ _ = Just
instance GLogical U1 U1 where gunify _ _ _ = Just
```

For the sum of types, we check if the same alternative is present in the two input values. If they are the same, then we recursively descend, otherwise, unification fails:

```
instance (GLogical f f', GLogical g g')
  => GLogical (f :+: g) (f' :+: g') where
  gunify _ (L1 x) (L1 y) = gunify (Proxy @f) x y
  gunify _ (R1 x) (R1 y) = gunify (Proxy @g) x y
  gunify _ _ _          = const Nothing
```

For products, we start by unifying first components, and then, if successful, we unify the second components:

```
instance (GLogical f f', GLogical g g')
  => GLogical (f **: g) (f' **: g') where
  gunify _ (x1 **: y1) (x2 **: y2) state = do
    state' <- gunify (Proxy @f) x1 x2
    gunify (Proxy @g) y1 y2 state'
```

When encountering a field of type `c` in the regular type, we expect `Term c` in its logical counterpart, and appeal to unification for `c`:

```
instance (Logical c)
  => GLogical (K1 i c) (K1 i' (Term c)) where
  gunify _ (K1 x) (K1 y) = unify' x y
```

Note that none of the instances above explicitly deal with the unification variables. This is handled by the `unify'` function that is defined in the core for `Term` and follows the standard MINIKANREN implementation for unification [4, 7]. We provide the implementation here for context:

```
unify' :: Logical a => Term a -> Term a -> State -> Maybe State
unify' l r state =
  case (shallowWalk state l, shallowWalk state r) of
    (Var x, Var y)
      | x == y -> Just state
    (Var x, r')
      | occursCheck' x r' state -> Nothing
      | otherwise -> addSubst x r' state
    (l', Var y)
      | occursCheck' y l' state -> Nothing
      | otherwise -> addSubst y l' state
    (Value l', Value r') -> unify l' r' state
```

Finally, we provide an instance for `M1` where we simply skip metadata:

```
instance (GLogical f f')
  => GLogical (M1 i t f) (M1 i' t' f') where
  gunify _ (M1 x) (M1 y) = gunify (Proxy @f) x y
```

With the generic instances in place, we provide an implementation for any type that has a generic representation:

```
genericUnify
  :: forall a.
    (Generic (Logic a), GLogical (Rep a) (Rep (Logic a)))
  => Logic a
  -> Logic a
  -> State
  -> Maybe State
genericUnify l r = gunify (Proxy @(Rep a)) (from l) (from r)
```

	exp, ms	log, ms	quines, ms	twines, ms	thrines, ms
<a href="#">faster-minikanren</a> [2] (Racket)	106.5	17.6	294.8	261.4	487.0
<a href="#">OCANREN</a> [13]	463.7	61.4	777.3	717.6	1 258.9
<a href="#">KLOGIC</a> [9]	942.2	87.8	1 071.6	1 260.5	5 071.0
TYPEDKANREN (this paper)	588.1	68.1	1 084.0	1 580.0	3 775.0

Fig. 1. Preliminary benchmark results. The benchmarks were executed on an ASUS ZenBook 15 with an AMD Ryzen 7 7735U CPU, 16GB of memory, using NixOS, Linux 6.10.2.

The implementation of [genericWalk](#), [genericOccursCheck](#), [genericInject](#), and [genericExtract](#) follows the same pattern. Complete implementation is available in the [Kanren.GenericLogical](#) module.

While generic implementation works well in most situations, there are some limitations, since not all Haskell data types provide a generic implementation. In particular, GHC cannot derive [Generic](#) instance for some fairly simple generalized algebraic data types (GADTs). While there do exist approaches to generic programming that works better with GADTs [24], they are not part of the standard [GHC.Generics](#) toolkit, so we leave such support for future work.

## 5 PERFORMANCE EVALUATION

To evaluate the performance of TYPEDKANREN, we have implemented

- (1) A relational arithmetic system, based on a binary representation [10]. The system includes the exponentiation and integer logarithm relations, which are used in the benchmarks. The implementation is available in the module [Kanren.Data.Binary](#).
- (2) A relational Scheme interpreter [4]. Running the interpreter “backwards” allows systematically searching for programs (S-expressions) that produce the desired result. In particular, such interpreter can be used to generate quines — programs that evaluate to themselves. Quine generation is used in the benchmarks. The implementation is available in the module [Kanren.Data.Scheme](#).

Preliminary benchmarks are shown in Fig. 1:

- (1) [exp](#) — computing  $3^5$  using the relational arithmetic system;
- (2) [log](#) — computing the integer logarithm  $\log_3 243$  using the relational arithmetic system;
- (3) [quines](#) — generation of 100 quines via the relational Scheme interpreter;
- (4) [twines](#) — generation of 15 twines<sup>17</sup> via the relational Scheme interpreter;
- (5) [thrines](#) — generation of 2 thrines<sup>18</sup> via the relational Scheme interpreter.

To benchmark the Haskell implementation, we use [criterion](#) [20], a widely-used Haskell library for performance measurement and analysis of Haskell functions. To ensure proper computation we force evaluation of results to normal form (via [NFData](#) instances). For benchmarks of other implementations, we reuse the setup from [KLOGIC](#) paper [9]. The code and instructions for replicating the benchmarks for [faster-minikanren](#), [OCANREN](#), [KLOGIC](#), and TYPEDKANREN is available at [github.com/SnejUgal/typedKanren-benchmarks](https://github.com/SnejUgal/typedKanren-benchmarks).

Based on the benchmark results, we can say that TYPEDKANREN performs on par with [KLOGIC](#), but at the moment is outperformed by all other implementations. We believe that the following factors contribute to this result:

<sup>17</sup>A *twine* is a pair of programs A and B such that A evaluates to B and B evaluates to A.

<sup>18</sup>A *thrine* is a triple of programs A, B, and C such that A evaluates to B, B evaluates to C, and C evaluates to A.



- (1) A feature that is shared by LISP-based implementations like `faster-minikanren` is cheap `inject`. In fact, in LISP all user values are essentially S-expressions and converting a value into one available for relational setting is not necessary. Similarly, `OCANREN` implements tagless logical values [13, §6.2], also allowing for zero-cost injection. This relies on polymorphic unification [13, §5], which might be possible in Haskell via another kind of generic programming [18].
- (2) We do not (yet) implement the `set-var-val!` optimization to reduce the cost of looking up variables. This appears to be one of the main optimizations in `faster-minikanren` [2] which at the moment is de facto the fastest implementation. While mutable variables are normally inaccessible in Haskell, we may use the local state [14] safely inside the `Goal` for unification variables to gain performance. A similar optimization is implemented in `unification-fd` library [23].  
That said, we have conducted some experiments implementing a version of this optimization both with `STRef`<sup>19</sup> and `IORef`<sup>20</sup>, none of which affected the performance of `TYPEDKANREN` positively.
- (3) At the moment we rarely use strictness annotations, and thus our implementation may accumulate too many unwanted thunks. We should perform proper strictness analysis to enhance performance and avoid excessive memory usage.
- (4) To ensure that the terms are fully evaluated, we rely on `NFData`. However, this might have a negative effect on the performance in the cases where the term is already fully evaluated. Although we do not believe that this contributes much, we should properly analyze the effect of forcing in the future.

## 6 CONCLUSION AND FUTURE WORK

We present a working embedding of relational programming in a typed non-strict functional programming language Haskell. Our implementation, `TYPEDKANREN`, is a feature full dialect of `MINIKANREN` with unification, disequality constraints, and static typing support. `TYPEDKANREN` is capable of expressing many classical relational programs, including relational arithmetic systems [10] and a relational Scheme interpreter [4], which we implement in full and use in our benchmarks.

To provide relational matching, we rely on prisms, an implementation of first-class patterns in Haskell. In addition to the typed version of `match`<sup>e</sup>, we discuss potential shortcomings of relational matching and provide a matching operator with exhaustiveness checking.

To assist the user with development of relational programs in presence of user-defined types, we provide generic programming and metaprogramming tools to generate logical counterparts to Haskell types and derive necessary instances to provide unification capabilities. For relational matching, we rely on existing metaprogramming tools provided by the `lens` library [12] to generate regular prisms and also provide a Template Haskell function to generate exhaustive prisms for logical types.

Unlike `OCANREN` [13], we do not provide any quasiquotation support to allow nicer syntax for relational programs. The main reason is that `TYPEDKANREN` is an active work-in-progress, and we prefer to exhaust design choices within Haskell syntax before leaning on quasiquotation with a separate parser. That said, support for quasiquotation is a prominent direction for future work.

<sup>19</sup>see [github.com/snejugal/typedKanren/pull/13](https://github.com/snejugal/typedKanren/pull/13)

<sup>20</sup>see [github.com/snejugal/typedKanren/pull/14](https://github.com/snejugal/typedKanren/pull/14)

Another useful form of automation is conversion of existing functional programs into relational ones [16]. Indeed, using Template Haskell it appears feasible to perform such transformations automatically for many Haskell functions.

On the performance side, TYPEDKANREN at the moment is underperforming (although not critically) compared with `faster-minikanren` (the fastest known untyped implementation), `OCANREN`, and `KLOGIC`. There are a number of well-known optimizations that should be properly implemented in the core of TYPEDKANREN, achieving performance that is on par or exceeds the competitive implementations. The first contender for a significant improvement is the Haskell analog of `set-var-val!` optimization. Other ideas for optimizations and heuristics speeding up substitution may be taken from Wren Romano’s `unification-fd` library.

## ACKNOWLEDGMENTS

We thank Nikolay Shilov for his comments on an earlier draft of this paper. We thank the anonymous reviewers of miniKanren’24 workshop for their valuable feedback on an earlier draft of this paper.

## REFERENCES

- [1] F. Baader and W. Snyder. 2001. Unification Theory. In *Handbook of Automated Reasoning*, J.A. Robinson and A. Voronkov (Eds.). Vol. I. Elsevier Science Publishers, 447–533.
- [2] Michael Ballantyne. 2015. *A fast implementation of miniKanren with disequality and absento, compatible with Racket and Chez*. <https://github.com/michaelballantyne/faster-minikanren>
- [3] William E. Byrd. 2009. *Relational Programming in miniKanren: Techniques, Applications, and Implementations*. Ph.D. Dissertation. USA. Advisor(s) Friedman, Daniel P. AAI3380156.
- [4] William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming* (Copenhagen, Denmark) (Scheme ’12). Association for Computing Machinery, New York, NY, USA, 8–29. <https://doi.org/10.1145/2661103.2661105>
- [5] Gabriella Gonzalez. [n.d.]. *total library*. <https://hackage.haskell.org/package/total>
- [6] Gabriella Gonzalez. 2015. *total-1.0.0: Exhaustive pattern matching using traversals, prisms, and lenses*. <https://www.haskellforall.com/2015/01/total-100-exhaustive-pattern-matching.html>
- [7] Jason Hemann and Daniel P. Friedman. 2013.  $\mu$ Kanren: 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>
- [8] Ralf Hinze and Simon Peyton Jones. 2001. Derivable Type Classes. *Electronic Notes in Theoretical Computer Science* 41, 1 (2001), 5–35. [https://doi.org/10.1016/S1571-0661\(05\)80542-0](https://doi.org/10.1016/S1571-0661(05)80542-0) 2000 ACM SIGPLAN Haskell Workshop (Satellite Event of PLI 2000).
- [9] Yury Kamenev, Dmitrii Kosarev, Dmitry Ivanov, Denis Fokin, and Dmitri Boulytchev. 2023. klogic: miniKanren in Kotlin. In *Proceedings of the Fifth miniKanren Workshop (miniKanren ’23)*. <http://minikanren.org/workshop/2023/minikanren23-final4.pdf>
- [10] Oleg Kiselyov, William E. Byrd, Daniel P. Friedman, and Chung-chieh Shan. 2008. Pure, Declarative, and Constructive Arithmetic Relations (Declarative Pearl). In *Functional and Logic Programming*, Jacques Garrigue and Manuel V. Hermenegildo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 64–80.
- [11] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, Interleaving, and Terminating Monad Transformers (Functional Pearl). In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming* (Tallinn, Estonia) (ICFP ’05). Association for Computing Machinery, New York, NY, USA, 192–203. <https://doi.org/10.1145/1086365.1086390>
- [12] Edward Kmett. [n.d.]. *lens library*. <https://hackage.haskell.org/package/lens>
- [13] Dmitrii Kosarev and Dmitry Boulytchev. 2018. Typed embedding of a relational language in OCaml. *arXiv preprint arXiv:1805.11006* (2018).
- [14] John Launchbury and Simon L. Peyton Jones. 1994. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) (PLDI ’94). Association for Computing Machinery, New York, NY, USA, 24–35. <https://doi.org/10.1145/178243.178246>
- [15] Petr Lozov and Dmitry Boulytchev. 2020. On Fair Relational Conjunction. *Proceedings of the 2020 miniKanren and Relational Programming Workshop* (2020), 1–12. <http://minikanren.org/workshop/2020/minikanren-2020-paper1.pdf>

- [16] Petr Lozov, Andrei Vyatkin, and Dmitry Boulytchev. 2018. Typed Relational Conversion. In *Trends in Functional Programming*, Meng Wang and Scott Owens (Eds.). Springer International Publishing, Cham, 39–58.
- [17] Kuang-Chen Lu, Weixi Ma, and Daniel P Friedman. 2019. Towards a miniKanren with fair search strategies. In *Proceedings of the 2019 miniKanren and Relational Programming Workshop*. 1–15. <http://minikanren.org/workshop/2019/minikanren19-final1.pdf>
- [18] Ralf Lämmel and Simon Peyton Jones. 2003. Scrap your boilerplate: a practical approach to generic programming. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'03)* (acm sigplan international workshop on types in language design and implementation (tldi'03) ed.). ACM Press, 26–37. <https://www.microsoft.com/en-us/research/publication/scrap-your-boilerplate-a-practical-approach-to-generic-programming/>
- [19] Chris Okasaki and Andrew Gill. 1998. Fast Mergeable Integer Maps. In *Workshop on ML*. 77–86. <https://git.sr.ht/~wklew/containers/blob/b4074eaabf2c2c0f87b0a096a7b4eb3a2f9dee97/papers/Okasaki%20and%20Gill%20-%201998%20-%20Fast%20Mergeable%20Integer%20Maps.pdf>
- [20] Bryan O'Sullivan. [n. d.]. criterion: *Robust, reliable performance measurement and analysis*. <https://hackage.haskell.org/package/criterion>
- [21] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The Reasoned Schemer*. MIT Press. <https://doi.org/10.7551/mitpress/5801.001.0001>
- [22] Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. 2017. Profunctor Optics: Modular Data Accessors. *Art Sci. Eng. Program.* 1, 2 (2017), 7. <https://doi.org/10.22152/PROGRAMMING-JOURNAL.ORG/2017/1/7>
- [23] Wren Romano. [n. d.]. unification-fd: *Simple generic unification algorithms*. <https://hackage.haskell.org/package/unification-fd>
- [24] Alejandro Serrano and Victor Cacciari Miraldo. 2018. Generic programming of all kinds. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) (*Haskell 2018*). Association for Computing Machinery, New York, NY, USA, 41–54. <https://doi.org/10.1145/3242744.3242745>
- [25] Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) (*Haskell '02*). Association for Computing Machinery, New York, NY, USA, 1–16. <https://doi.org/10.1145/581690.581691>
- [26] Erik Simmler. [n. d.]. canrun\_rs: *a Rust logic programming library inspired by the \*Kanren family of language DSLs*. [https://github.com/tgecho/canrun\\_rs](https://github.com/tgecho/canrun_rs)
- [27] Twan van Laarhoven. 2009. CPS based functional references. <https://www.twanvl.nl/blog/haskell/cps-functional-references>
- [28] Philip Wadler. 1990. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France) (*LFP '90*). Association for Computing Machinery, New York, NY, USA, 61–78. <https://doi.org/10.1145/91556.91592>