NASA/TM–20240010266

Logic Programming with Extensible Types

Ivan Perez
KBR @ NASA Ames Research Center, Moffett Field, CA, USA

Angel Herranz
Universidad Politecnica de Madrid, Boadilla del Monte, Madrid, Spain

August 2024

NASA/TM–20240010266

Logic Programming with Extensible Types

Ivan Perez
KBR @ NASA Ames Research Center, Moffett Field, CA, USA

Angel Herranz
Universidad Politecnica de Madrid, Boadilla del Monte, Madrid, Spain

National Aeronautics and
Space Administration

Ames Research Center, Moffett Field, CA 94035

August 2024

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500

Abstract

Logic programming allows structuring code in terms of predicates or relations, rather than functions. Although logic programming languages present advantages in terms of declarativeness and conciseness, the introduction of static types has not become part of most popular logic programming languages, increasing the difficulty of testing and debugging of logic programming code. This paper demonstrates how to implement logic programming in Haskell, thus empowering logic programs with types, and functional programs with relations or predicates. We do so by combining three ideas. First, we use extensible types to generalize a type by a parameter type function. Second, we use a sum type as an argument to introduce optional variables in extensible types. Third, we implement a unification algorithm capable of working with any data structure, provided that certain operations are implemented for the given type. We demonstrate our proposal via a series of increasingly complex examples inspired by educational texts in logic programming, and leverage the host language's features to make new notation convenient for users, showing that the proposed approach is not just technically possible but also practical.

Contents

1 Introduction

Definitions in functional languages are structured around functions, which are executed by providing the inputs and evaluating the result. For example, given the type `data Nat = Zero | Suc Nat`, we can define addition as:

```
plus :: Nat -> Nat -> Nat
plus Zero    y = y
plus (Suc x) y = Suc (plus x y)
```

Although, at an abstract level, functions are relations between sets, it is not usually possible to treat functions as relations in a language like Haskell. For example, we cannot use `plus` to efficiently calculate the subtraction function or to generally calculate all tuples of inputs and outputs to a function that meets a certain condition.

Contrast this limitation with how one would write an analogous *function* in a logic programming language like Prolog. We could define a predicate `plus` as:

```
plus(zero,  Y, Y).
plus(suc(X),Y, suc(Z)) :- plus(X,Y,Z).
```

In Prolog, `plus(A,B,C)` is true if `C` represents the sum of `A` and `B`. The predicate can be used to add two numbers, subtract a number from another, or check if two numbers add up to a given third, or obtain tuples of inputs and output for which the relation holds:[1]

```
?- plus(suc(suc(zero)),B,suc(suc(suc(zero)))).
B = suc(zero).
?- plus(zero,zero,suc(zero)).
false.
?- plus(A,suc(zero),C).
A = zero, C = suc(zero);
A = suc(zero), C = suc(suc(zero)).
```

This paper describes how we can write predicates in the style of logic programming in Haskell. To that end, we use extensible types, an approach to parameterize type definitions by a type function that is applied to every element inside a type's definition (Perez, 2023). We show, by means of example, that the proposed approach requires very little work on the side of the programmer, can capture many of the use cases of logic programming languages, and can be enabled by convenient notation.

The benefit of embedding logic programming in a typed functional language is two-fold. First, using logic programming can reduce code duplication and enable more declarative specifications. Second, it enables using static typing in logic programs, a feature that has eluded most popular logic programming languages.

Specifically, the contributions of this paper are:

[1]We manually stop the production of solutions to the last query.

- We present a simple interface for logic programming that facilitates introducing logic variables in algebraic datatypes and expressing unification constraints (Section 3).
- We show that the proposed approach is applicable to polymorphic types, and enables leveraging the host language's mechanisms for type inference and higher-order to implement type-safe higher-order logic programming (Section 4).
- We extend the language with *cuts*, which allow users to increase performance, provide determinism, and encode negation as failure (Section 5).
- We exemplify the benefits of our approach with additional examples that combine functional constructs and logic programming (Section 6).

We discuss our implementation in Section 7 and limitations in Section 8. Section 9 details related work, and Section 10 proposes future work.

2 Background

To make this paper sufficiently self-contained, we briefly introduce extensible types (Perez, 2023). Readers familiar with extensible types can skip to Section 3.

The extensible type design pattern is a pattern in which the definition of an algebraic data type is parameterized by a type function that is applied to every element of the definition. For example, given a type representing expressions in some language:

```
data Expr = Const Double
          | Add Expr Expr
          | Neg Expr
```

we define the corresponding extensible type as follows:

```
data ExprF f = ConstF (f Double)
             | AddF (f (ExprF f)) (f (ExprF f))
             | NegF (f (ExprF f))
```

If we use the polymorphic type `Identity` as type function `f`, the resulting representation is isomorphic to the original `Expr`. Other parametric types and type functions render different results. For example, a type that pairs elements with a tuple of `Int`s can be used to annotate values with the line and column where they were found in an input file, which is useful in compilers to report error information to the user. Applying a `Maybe` or `Either`, an extensible type makes every element optional. That representation is also useful for parsing, to represent that a branch of the abstract syntax tree (AST) failed to parse while recovering from the parsing error and continuing to parse the rest of the input.

Other polymorphic types and type functions enable changing type definitions to introduce new cases, prune branches, replace elements, etc. Type functions can also be composed; when applied to ASTs, the composition of extensible types can capture language embeddings.

The application of a specific type function to an extensible type does not determine how it should be interpreted. For example, `Either String` can be used to annotate failed AST branches with the reasons why values could not be parsed from an input file, but also to replace branches in the AST by variables with the given variable names. This idea will be used in future sections to replace portions of a datatype with variables in predicate definitions and logic programming queries.

In the rest of the text, we refer to types that are parameterized in this manner simply as *extensible types*. Other approaches to parameterize a type by a type function used in its definition are further discussed in Section 9.

3 Logic Programming with Extensible Types

This section introduces primitives to define and combine predicates, and ways to capture relations between values of algebraic datatypes. We first introduce basic types and a few simple primitives and connectives. We later show how to introduce logic variables in types, to replace portions of values with logic variables, and how to express relations involving types with variables.

3.1 Predicates, Primitives and Boolean Combinators

The elementary type in our proposal is a `Predicate`, which denotes a logic predicate. We keep the type abstract for now and discuss implementation details later.

To interact with predicates, we provide the function `repl :: Predicate -> IO ()` that, when applied to a `Predicate`, produces possible solutions one by one, similar to how one interacts with the REPL of a logic programming language. If any constraints on variables apply for the predicate to hold, `repl` prints the constraints. Otherwise, the function only prints "`true.`" or "`false.`". Note that, when using the REPL to illustrate examples in the paper, we often align the queries for readability.

3.1.1 Primitives

We provide `succeed :: Predicate`, which holds without additional constraints, and its counterpart, `fail :: Predicate`.

Example We evaluate predicates a session as follows:

```
> repl fail
false.
```

□

3.1.2 Boolean Connectives

Our counterparts to the boolean connectives *and* and *or*, which we denote `(@@)` and `(@|)`, allow users to combine predicates:

```
(@@) :: Predicate -> Predicate -> Predicate
(@|) :: Predicate -> Predicate -> Predicate
```

Example The following queries show how we can use the boolean connectives to combine `succeed` and `fail`. The results should be straightforward:

```
> repl (succeed @@ fail)
false.
> repl (succeed @@ (fail @| succeed))
true.
```

□

3.2 Terms and Logic Variables

To introduce logic variables in values, we are going to apply a type function to extensible types. We introduce a custom sum type `Term`, which can represent a logic variable with a name, or an actual value of a given type:[2]

```
data Term a = Var String | Compound a
```

When `Term` is applied to an extensible type, every element inside the latter can potentially be replaced with a logic variable, allowing us to describe values in which some portions are concrete and some portions are not.

Example Given the usual encoding of Peano numbers using a data type defined as `data Nat = Zero | Suc Nat`, the equivalent extensible type in Haskell would be:

```
data NatF f = ZeroF | SucF (f (NatF f))
```

We can use `NatF Term` to represent a natural number where part of the definition is substituted by a variable. To make the complete number replaceable with a variable, we enclose the type inside an additional `Term`:

```
type NatTerm = Term (NatF Term)
```

Examples of values of type `NatTerm` include `Var "y"`, representing a natural number denoted by the variable `"y"`, `Compound (SucF (Var "x"))`, representing the successor of `"x"`, and `Compound (SucF (Compound ZeroF))`, representing 1. In Prolog, such terms could be encoded as `suc(zero)`, `suc(X)`, and `Y`, respectively.

□

3.3 Term Unification

We have designed a domain-specific language to write predicates on types for which we can perform term unification and variable substitution. To unify two terms, we provide `(=:=) :: Term a -> Term a -> Predicate`.[3]

[2]In logic programming, a term can be a variable, or a compound term composed of an identifier and its arguments, which are also terms.

[3]The function is not fully polymorphic; there are constraints applicable to the parameter `a`, which we detail in Section 7.

Example We can define the successor predicate as:[4]

```
isSuc :: NatTerm -> NatTerm -> Predicate
isSuc x y = Compound (SucF x) =:= y
```

We can now query this predicate in a session, for example, to check if it holds for two *ground* values:[5]

```
> repl $ isSuc
    (Compound (SucF (Compound ZeroF)))
    (Compound (SucF (Compound
                      (SucF (Compound ZeroF)))))
true.
> repl $ isSuc (Compound (SucF (Compound ZeroF)))
               (Compound (SucF (Compound ZeroF)))
false.
```

The real power of our approach is that we can now use variables to provide one value and "obtain" the other:

```
> repl $ isSuc
    (Compound (SucF (Compound ZeroF))) (Var "x")
x = Compound (SucF
      (Compound (SucF (Compound ZeroF)))).
> repl $ isSuc (Var "x")
               (Compound (SucF (Compound ZeroF)))
x = Compound ZeroF.
```

The `repl` function prints the *answer substitutions.*

□

3.4 Existential Quantification

To introduce free variables in the *body* of predicates, akin to introducing free variables in the antecedent in predicate definitions in logic programming languages, we define `exists`:

```
exists :: (Term a -> Predicate) -> Predicate
```

When the function `exists` is applied to an argument predicate, it ensures that the variable provided to the given predicate is free.

Example Using all the definitions provided so far, we can implement the predicate `leq` (i.e., less than or equal to) to compare two natural numbers, `x` and `y`. The first rule of the comparison is that, if the first number `x` is zero, then `leq x y` must necessarily hold as there is no smaller number, that is, `x =:= Compound ZeroF`. The

[4]Compare with the Prolog program `is_suc(X,Y) :- suc(X) = Y.`

[5]We use *ground* to refer to *terms* that do not contain variables. The word ground also has meaning when discussing data types and generic programming, but we use the word exclusively with the former meaning.

second rule is that, if both elements are successors of other elements, respectively `x'` and `y'`, then the predicate holds if it holds for `x'` and `y'`:

```
exists $ \x' -> exists $ \y' ->
    x =:= Compound (SucF x')
 @@ y =:= Compound (SucF y')
 @@ leq x' y'
```

Combining both rules we obtain:

```
leq :: NatTerm -> NatTerm -> Predicate
leq x y = x =:= Compound ZeroF
       @| ( exists $ \x' -> exists $ \y' ->
               x =:= Compound (SucF x')
            @@ y =:= Compound (SucF y')
            @@ leq x' y' )
```

Moving the introduction of free variables to the top of the definition helps group all rules together, which can sometimes aid readability:

```
leq :: NatTerm -> NatTerm -> Predicate
leq x y = exists $ \x' -> exists $ \y' ->
     x =:= Compound ZeroF
  @| x =:= Compound (SucF x')
     @@ y =:= Compound (SucF y')
     @@ leq x' y'
```

As illustrated above, predicates can be *recursive*.

□

3.5 Notation

The principles of our proposal are applicable to languages with higher-kinded polymorphism. However, Haskell's ability to overload notation and define operators can make logic programming more convenient.

Specifically, Haskell allows us to define new operators and adjust their associativities and priorities. In our case, `(@@)` binds more strongly than `(@|)`, and `(=:=)` binds more strongly than either of them. We define synonyms `C` and `V` that match, respectively, `Compound` and `Var`, and pattern synonyms `Zero = C ZeroF` and `Suc x y = C (SucF x y)`. Using these facilities, `leq` can now be defined more succinctly as follows:

```
leq :: NatTerm -> NatTerm -> Predicate
leq x y = exists $ \x' -> exists $ \y' ->
     x =:= Zero
  @| x =:= Suc x' @@ y =:= Suc y' @@ leq x' y'
```

To help the reader understanding our approach, we refrain from relying too heavily on syntactic sugar during this exposition. Our implementation provides aids to make using logic programming more convenient, such as the ability to use strings to mean `Var` applied to such strings, and to use numbers to refer to ground `NatTerms`. This is further detailed in Section 7.

4 Polymorphism and Higher Order

The ability to write predicates using the approach described so far extends also to polymorphic types. Let us demonstrate with the types of polymorphic lists, frequently used both in logic programs and in functional programs. To extend a standard list type definition with an extra type function, we define:

```
data ListF f a = NilF | ConsF (f a) (f (ListF f a))
```

We can make it possible for any portion of a list, be it an individual element or the tail of the list, to be a variable by apply the `Term` type function:

```
type ListTerm a = Term (ListF Term a)
```

Example We can combine `ListTerm` with `NatF` to represent lists of natural numbers:

```
type NatListTerm = ListTerm (NatF Term)
```

The following term encodes the list `[0, 1, 2]`:

```
l1 :: NatListTerm
l1 = C $ ConsF (C ZeroF)
   $ C $ ConsF (C (SucF (C ZeroF)))
   $ C $ ConsF (C (SucF (C (SucF (C ZeroF))))) Nil
```

The following term encodes the list that starts with a 1, and whose tail is represented by a variable `"tl"`:

```
l2 :: NatListTerm
l2 = C (ConsF (C (Suc (C Zero))) (V "tl"))
```

□

Notation Like before, we introduce two pattern synonyms to simplify writing terms of type `ListTerm`: `Cons x y = C (ConsF x y)` and `Nil = C NilF`. Using all pattern synonyms we have defined so far, the list `l1` in the previous example could also be defined as:

```
l1 :: NatListTerm
l1 = Cons Zero $ Cons (Suc Zero)
   $ Cons (Suc (Suc Zero)) Nil
```

Our implementation allows us to write expressions like `[0, "x"]` to mean a list with two elements, where the first is zero and the second is the variable `"x"`. To help the reader gain intuition about how our approach works, we delay notation aids to Section 7.

4.1 Polymorphism and Type Safety

Using extensible types does not prevent the host programming language from performing type checking, including for types whose non-extensible variants were already polymorphic. If we try to use a term with the wrong type in the definition of a predicate or in a unification constraint, the type checker will be able to indicate it just like it would with any other type error.

Example Take the following predicate that checks whether an element is the head of a list (we avoid calling the function `head` to avoid name clashes):

```
isHead :: ListTerm a -> Term a -> Predicate
isHead x y = exists $ \v -> x =:= Cons y v
```

If we try to call `isHead` with a second argument of the wrong type, GHC's type checker will warn us:

```
> repl $ isHead l1 ((Var "x") :: Term Bool)
<interactive>:9:19: error:
      Couldn't match type 'Bool' with 'NatF Term'
      Expected type: Term (NatF Term)
        Actual type: Term Bool
```

□

This level of safety is especially important in order to write correct code that involves free variables. Relying on a host language with strong, static types means that the type checker can ensure that we are using variables in type-consistent ways.

Example We can check if a value is a member of a list by checking against the head or recursing into the tail. We do not have a case for the empty list: in that case, the predicate will fail because no rule will apply.

```
member :: Term a -> List a -> Predicate
member x xs =
     ( exists $ \tl -> xs =:= Cons x tl )
  @| ( exists $ \hd -> exists $ \tl ->
         xs =:= Cons hd tl @@ member x tl )
```

The free variable `hd` in the second rule has type `Term a`, and `tl` has type `List a`. The compiler is able to deduce this automatically because both are used as arguments to `Cons` and the resulting term unified with `xs`, whose type is known. If, for example, we introduce a condition `xs =:= hd`, or `member tl tl`, the compiler will be able to tell that we are using variables in inconsistent ways.

□

As expected, the type checker can infer the types of terms if we apply operations to them. For example, if we apply a predicate on `NatTerm` to the elements of the list, the type checker infers that all elements are naturals.

Example Let us demonstrate by implementing a predicate that checks whether all elements in a list are sorted. The predicate holds trivially for lists of zero or one elements; if there are more, we check the first two and recurse into the tail of the list: To compare elements, we use `leq :: NatTerm -> NatTerm -> Predicate`:

```
sorted v =  v =:= Nil
        @| ( exists $ \e1 -> v =:= Cons e1 Nil)
```

```
         @| ( exists $ \e1 ->
              exists $ \e2 ->
              exists $ \ts ->
                 v =:= Cons e1 (Cons e2 ts)
              @@ leq e1 e2
              @@ sorted (Cons e2 ts) )
```

GHC correctly infers that `v` has type `Term (ListF Term (NatF Term))`. □

4.2 Higher-order Logic Programming

Haskell's support for first-class functions immediately empowers our proposal for logic programming with higher-order. For logic programs, this means that we can pass predicates as arguments to other predicates.

Example Let us illustrate with a generalized version of `sorted` that takes a comparison *predicate* as argument:

```
sortedWith :: (Term a -> Term a -> Predicate)
           -> ListTerm a
           -> Predicate
sortedWith compare v =  v =:= Nil
         @| ( exists $ \x -> v =:= Cons x Nil )
         @| ( exists $ \x1 -> exists $ \x2 ->
              exists $ \xs ->
                 v =:= Cons x1 (Cons x2 xs)
              @@ compare x1 x2
              @@ sortedWith compare (Cons x2 xs) )
```

□

Because Haskell is strongly and statically typed, it provides a level of type safety that goes beyond what languages like Prolog offer, since they cannot assure that the types match without additional, hand-coded runtime checks. In general, calling a Prolog predicate with arguments of the wrong types will return `false`, just as if the predicate did not hold for those inputs (because it does not!), making this kind of type error hard to identify. We further expand on this idea in Section 7.

We can use the same approach to generalize standard functions and turn them into predicates, such as the Haskell list function `map` that applies a transformation to each element in a list.

Example The following program defines a predicate analogous to the `Prelude` function `map`:

```
mapP :: (Term a -> Term b -> Predicate)
     -> ListTerm a -> ListTerm b -> Predicate
mapP f l1 l2 =
     l1 =:= Nil @@ l2 =:= Nil
  @| ( exists $ \l10 -> exists $ \l1s ->
       exists $ \l20 -> exists $ \l2s ->
          l1 =:= Cons l10 l1s
       @@ l2 =:= Cons l20 l2s
       @@ f l10 l20
       @@ mapP f l1s l2s )
```

Let us demonstrate the use of `mapP`. Given a predicate `isSuc`, which pairs each number with its successor, we can use is to add 1 to every element of a list:

```
listPlusOne :: ListTerm (NatF Term)
            -> ListTerm (NatF Term)
            -> Predicate
listPlusOne = mapP isSuc
```

Like before, Haskell's type checker ensures that the predicate that we pass to `mapP` has type `Term (NatF Term) -> Term (NatF Term) -> Predicate`, removing the possibility of type errors at runtime.

□

5 Cuts

Cuts are used in logic programming to limit the application of backtracking to search for alternative solutions. Consider the following (incorrect) implementation of the remainder algorithm:

```
remainder :: NatTerm -> NatTerm -> NatTerm
          -> Predicate
remainder n q r =
     lt n q @@ n =:= r
  @| exists $ \diff ->
       plus q diff n @@ remainder diff q r
```

This definition will never work if `q` is zero, since the first rule will fail, and the execution of the second rule will lead the program into an infinite loop. We could try to protect ourselves from such cases by introducing an additional case that fails if `q` is zero (`q =:= Zero @@ fail`). However, if we try to invoke this predicate with `Zero` as second argument, the evaluation will still try to backtrack, fail to show that `n` is smaller than `q`, and recurse using the third rule.

To prevent such cases, we introduce the functions `scope` and `(@!)`, which help control backtracking or the evaluation of alternatives. Inspired by the notion of cuts in Prolog, we refer to `(@!)` as our own cut operator. We can re-write `remainder` using the cut operator, and `scope` to limit the rules affected by the cut:

```
remainder :: NatTerm -> NatTerm -> NatTerm
          -> Predicate
remainder n q r =
  scope $ q =:= Zero @! fail
       @| lt n q @@ n =:= r
       @| (exists $ \diff ->
             plus q diff n @@ remainder diff q r)
```

Callers to `remainder` will be unaware that the predicate uses cuts internally in its implementation.

5.1 Negation as Failure

Cuts can be used to implement a form of negation, with:

```
neg :: Predicate -> Predicate
neg p = scope $ p @! fail @| succeed
```

Example It is frequently useful to state that two terms cannot unify, for which we define an operator `(=/=)` as:

```
(=/=) :: Term a -> Term a -> Predicate
(=/=) x y = neg (x =:= y)
```

Similarly, we can implement a predicate that holds only if a given term is *not* a member of a given list:

```
notMember :: Term a -> List a -> Predicate
notMember x xs = neg (member x xs)
```

This kind of negation, called *negation as failure*, is a weak form of negation. If either of the arguments of `(=/=)` is still a variable at the time when the unification algorithm tries to evaluate whether the predicate holds, the unification `x =:= y` will hold, making its negation fail. This makes some implementations by negation not work without additional aids. For example, one cannot use `notMember` as defined above to find possible values of a variable that are not members of a list. Instead, it is necessary to *ground* the term first. To this end, we provide the predicate `isGround` which, for any type for which ground terms have a finite representation, holds only if the given argument contains no variables.

6 Examples

We further exemplify our proposal with two examples: path search in graphs, and solving Sudoku puzzles.

6.1 Graphs

Let us imagine that we have a graph, and want to determine if there is a path between two vertices. In this example we shall represent a specific graph with a limited number of vertices. Our goal is not to provide a generic library for path searching using logic programming, but simply to encode how a graph can be represented and queried. For the rest of the section, we will use the graph depicted in Fig. 1 as an example.

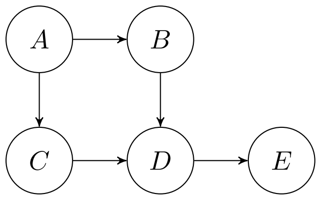

Figure 1: A directed graph with five vertices.

6.1.1 Vertices

We can represent the graph vertices by means of a specific enum type:

```
data Vertex = VertexA | VertexB | VertexC
            | VertexD | VertexE
```

The extensible type counterpart is defined as:[6]

```
data VertexF (f :: * -> *)
  = VertexAF | VertexBF | VertexCF | VertexDF
  | VertexEF
```

We also define the type synonym `VertexTerm = Term (VertexF Term)`, and the pattern synonyms `VertexA = C VertexAF`, `VertexB = C VertexBF`, and so on.

6.1.2 Graphs

To express the edges of the graph, we can use a dedicated predicate, which we shall call `edge`. For the sake of the example, let us define a predicate that encodes the graph depicted in Fig. 1.

```
edge :: VertexTerm -> VertexTerm -> Predicate
edge v1 v2 = v1 =:= VertexA @@ v2 =:= VertexB
          @| v1 =:= VertexA @@ v2 =:= VertexC
          @| v1 =:= VertexB @@ v2 =:= VertexD
          @| v1 =:= VertexC @@ v2 =:= VertexD
          @| v1 =:= VertexD @@ v2 =:= VertexE
```

6.1.3 Paths

As stated, we want to check if two vertices are connected in the graph. More specifically, we want to know the possible paths between two vertices. To represent paths, we use a list of vertices, that is:

```
type PathTerm = ListTerm (VertexF Term)
```

Now we can define the `path` predicate as follows:

[6]For enums, the extensible type is simply a polymorphic type that takes, and ignores, the higher-kinded type function argument. It is equally possible to simply use `Vertex` instead of `VertexF Term`, which would make for simpler code. In this case, we use the latter for the sake of regularity.

```
path :: VertexTerm -> VertexTerm -> PathTerm
     -> Predicate
path start end path = exists $ \revPath ->
     traverseP start end [start] revPath
  @@ reverseL revPath path
```

where `traverseP` is a predicate that holds if there is a path to go from a start vertex to an end vertex, without going through any vertex that has already been visited. The header of the function is:

```
traverseP :: VertexTerm -> VertexTerm
          -> PathTerm -> PathTerm -> Predicate
traverseP start end visited path =
```

If the path is formed by the end vertex followed by the visited vertices in reverse order, and there is a path from the current vertex `start` to `end`, the predicate holds:

```
path =:= Cons end visited @@ edge start end
```

The second rule uses negation: if there is a next step that is *not* the end vertex, has *not* been visited, and there is a path from it to the end vertex, the predicate holds:

```
@| ( exists $ \next ->
      edge start next
      @@ next =/= end
      @@ notMember next visited
      @@ traverseP next end (Cons next visited)
           path )
```

We can query this predicate in a terminal with `repl`:

```
> repl $ path VertexC VertexE "path"
path = c : d : e : [].
> repl $ path VertexA VertexE "path"
path = a : b : d : e : [] ;
path = a : c : d : e : [].
```

We can check that it does not find impossible paths:

```
> repl $ path VertexE VertexA "path"
false.
```

We can use more complicated queries, such as asking for any way to get to the vertex E, obtaining:

```
> repl $ path (V "start") VertexE (V "path")
start = d, path = d : e : [] ;
start = a, path = a : b : d : e : [] ;
start = a, path = a : c : d : e : [] ;
start = b, path = b : d : e : [] ;
start = c, path = c : d : e : [].
```

6.1.4 Generalizing Graph Search

The example above hard-codes the graph in the code, but we may want to use the same ideas with multiple graphs. We can parameterize `edge` by a list of edges:

```
edge :: [(VertexF Term, VertexF Term)]
     -> VertexF Term -> VertexF Term
     -> Predicate
edge xs v1 v2 = foldr f fail xs
  where
    f p (v1', v2') = p @| v1 =:= v1' @@ v2 =:= v2'
```

The predicates `path` and `traverseP` would have to be parameterized by the graph edges in the same way.

Note that nothing in the definition of `edge` that forces the inputs to be vertices and lists of vertices. If we remove the type signature, the compiler would infer that the function is well defined for any `Foldable` carrying pairs of elements of some `Term`, respectively with the second and third argument. This highlights that one can leverage the full power of Haskell when specifying predicates and writing functions that generate logic programs (i.e., meta-programming), including function composition, traversals, etc.

6.2 Sudoku

Finally, let us demonstrate how to implement a Sudoku solver. The purpose of this example is not to implement the most efficient Sudoku solver, but to demonstrate the simplicity and expressiveness of our approach. For that reason, we limit the size of the Sudoku to 4x4 puzzles, divided in 2x2 regions or blocks. This example also illustrates how it is possible to combine extensible and non-extensible types, which can sometimes be necessary or just more convenient.

First, we define an enum to represent digits. The extensible type variant is defined as follows:

```
data DF (f :: * -> *) = D1F | D2F | D3F | D4F
```

We introduce pattern synonyms `P1 = C D1F`, `P2 = C D2F`, etc. and the type synonym `PTerm = Term (DF Term)`. We use the encoding of lists described earlier to represent grids as `Term (ListF Term (DF Term))`.

A Sudoku is correct if no numbers repeat in any individual row, column, or block. For example, if we assign the variables `v11` to the top left element of the grid, `v12` to the one to its right, and so on, then the constraints that would apply to that element would be that:

```
   v11 =/= v12 @@ v11 =/= v13 @@ v11 =/= v14
@@ v11 =/= v21 @@ v11 =/= v31 @@ v11 =/= v41
@@ v11 =/= v12 @@ v11 =/= v21 @@ v11 =/= v22
```

We can generalize those constraints for all elements of the grid as follows:

```
solved :: Term (ListF Term (DF Term)) -> Predicate
solved ls = exist 16 $ \rs ->
```

v11	**v12**	**v13**	**v14**
v21	**v22**	v23	v24
v31	v32	v33	v34
v41	v42	v43	v44

Figure 2: 4x4 Sudoku highlighting the cells whose values cannot coincide with the one in position $(1, 1)$

```
    ls =:= listOf rs
 @@ allP isGround rs
 @@ ( allP different $
            [ row    rs n     | n  <- [0..3] ]
         ++ [ column rs n     | n  <- [0..3] ]
         ++ [ block  rs n1 n2 | n1 <- [0..1]
                              , n2 <- [0..1] ] )
```

Our code uses four additional auxiliary functions:

- `exist :: Int -> ([Term a] -> Predicate) -> Predicate` is a variant of `exists` that generates multiple variables of the same type (i.e., type `[Term a]` for any `a`).
- `listOf :: [Term a] -> ListF Term a` turns a Haskell list into an extensible list.
- `allP :: (Term a -> Predicate) -> [Term a] -> Predicate` succeeds if the given predicate holds for every element of a list.
- `different :: [Term a] -> Predicate` holds if every element of a list is different from every other element. The function uses `(=/=)` to compare elements, we require that elements be bound to ground terms before comparing them, as explained earlier in this section.
- The functions `row`, `column` and `block` take a flattened 4x4 matrix and produce the elements in a specific row, column or 2x2 block, respectively.

We can use this predicate to both check existing complete Sudoku grids, as well as to search for solutions to incomplete grids. The time it takes to run depends on the number of clues or filled positions in the puzzle, going from milliseconds in the case of complete or almost complete grids to 1-2 seconds in the case of grids with few clues (when compiled). As an example, given the following list, which encodes the grid in Fig. 3.a:

1	2	”r1c3”	4
3	”r2c2”	1	”r2c4”
”r3c1”	”r3c2”	2	1
2	1	4	”r4c4”

(a) 4x4 Sudoku with several unsolved positions.

1	2	3	4
3	4	1	2
4	3	2	1
2	1	4	3

(b) Solved 4x4 Sudoku.

Figure 3: Unsolved and solved Sudoku puzzles.

```
ex :: Term (ListF Term (DF Term))
ex = [ P1,       P2,       V "r1c3", P4
     , P3,       V "r2c2", P1,       V "r2c4"
     , V "r3c1", V "r3c2", P2,       P1
     , P2,       P1,       P4,       V "r4c4" ]
```

our predicate resolves the values of the variables as follows (Fig. 2.b depicts the solution):

```
> repl $ solved ex
r1c3 = 3, r2c2 = 4, r2c4 = 2, r3c1 = 4, r3c2 = 3, r4c4 = 3.
```

7 Implementation

We have implemented the ideas in this paper in Haskell, including types representing `Term`s and `Predicate`s; predicate building functions and combinators; classes that define the operations that types must support for unification to be used on them; a unification algorithm; several execution functions to evaluate predicates. Overall, our implementation only needs 350 lines of code, without considering spaces or comments.

7.1 Types and Typeclasses

We define three key types: a polymorphic type `Term`, described in Section 3, and a polymorphic type `Tree`, representing a tree of constraints, and a type synonym `Predicate`, which represents a predicate.

To perform unification for terms of arbitrary types, we require that three operations be supported on them:

- The ability to unify two terms of that type. This requires determining if the constructors of both values are the same and call our unification recursively for the arguments to the constructor. We provide a definition of unification for `Term`.

- The ability to check if a variable is used in a term. This function takes a variable name and returns a boolean, and does not generally need to have any logic, but merely call our own occurs check function on all arguments to the type constructor.

- The ability to substitute a variable by a term inside another term. Again, we provide a function that performs substitution for values of type `Term a` (for some `a`), so users only need to call our function recursively on all arguments of their constructors.

We capture these operations via three type classes: `UnifyLocal`, `Occurs`, and `Substitute`. To shorten type signatures, we defined a type class `Logic` as a shorthand for types that are instances of all three classes.

7.2 Predicate Evaluation

To evaluate predicates, we provide several functions:

- `eval :: Predicate -> IO ()` pretty prints all solutions to a predicate (if any).

- `repl :: Predicate -> IO ()` tries to mimic the interface of a standard Prolog REPL by printing solutions one by one, letting users control the production of solutions with the keys space and enter.

- `isSatisfiable :: Predicate -> Bool` determines if a `Predicate` admits any solutions.

- `valueOf :: Logic a =>Term a -> Predicate -> Maybe (Term a)` provides a value for a given variable under which a `Predicate` may hold. It is a requirement that the argument `Term` be a variable.

- `findAll :: Logic a =>Term a -> Predicate -> [Term a]` provides all values for a given variable under which a `Predicate` may hold. It is a requirement that the argument `Term` be a variable.

The functions `eval` and `repl` make it easier to evaluate predicates from GHCi, while the other three make it easier to interact with these facilities from Haskell code.

Predicates represent constraints using a tree-like structure. At any point in the tree, we can find a leaf, or a branch with a unification constraint followed by several possibilities. Concatenation of predicates with `(@@)` (i.e., and) simply appends one tree at the end of all the leaves of the other tree. Combining predicates with `(@|)` creates a new branch that unifies trivially, followed by the two possible solutions, arguments to `(@|)`.

To evaluate solution, we traverse the solution tree to 1) replace unification constraints with more specific constraints on the subterms, and 2) substitute variables with their expected values in other terms.

When new branches are introduced, our datatype also expects the current state used to generate fresh variable names, and returns a new value for such state. This mechanism makes solution generation lazier, as our original implementation, which leveraged a state monad, did not work well for recursive predicates.

A final step flattens the solution tree, turning it into a list of lists of unification constraints that have been processed. The functions `eval` and `repl` traverse the list, eliminating trivial constraints that clutter the output.

7.3 Notation and Usability

To make our DSL more convenient, we have implemented facilities to make the notation succinct and familiar, and reduce how much code users must write.

We have implemented generic programming (Magalhães et al., 2010) aids to generate instances of the classes that our unification algorithm requires to operate on algebraic data types. For example, for `NatF`, users only need to declare:

```
deriving instance Eq (NatF Term)
instance UnifyLocal  (NatF Term)
instance Occurs      (NatF Term)
instance Substitute  (NatF Term)
instance Logic       (NatF Term)
```

Users can define their own `Show` functions if they prefer to print values in a more concise manner than Haskell's default. For example, to show natural numbers we define a custom `Show` instance such that ground terms are printed as actual numbers (e.g., `1`, `2`), and other numbers are printed as the addition of a number and a variable (e.g., `1 + x`, `55 + z`). We implement similar instances for `ListF`, so that values are printed in a familiar notation (e.g., `1 : x : 5 : xs`).

To make working with such instances easier, we also implement instances to overload numbers, strings and lists. For example, providing `1` when a `Term (NatF Term)` is expected denotes `C (SucF (C ZeroF))`. We overload strings such that providing a literal string where a `Term` is expected means the `V` (or `Var`) with such string as argument. This renders very concise expressions:

```
> plus 1 "x" 5
x = 4
```

Finally, we overload list syntax so that it is possible to provide lists elements using Haskell's standard notation for lists. For example, the following is fully supported:

```
> isTail [1, 2, 3] [2, 3]
true.
```

Using Haskell's support to overload syntax, it is possible to combine lists of ground terms with variables. For example, for lists of natural numbers, `[1, "x"]` is the same as `Cons (Suc Zero) (Cons (V "x") Nil)`. In our implementation, we also use the notation `hd :< tl` to mean the *term* `Cons hd tl`, where `hd` is an element term `hd` and `tl` is a list term.

The ability to provide such a convenient syntax is possible thanks to Haskell's abilities to overload standard syntax. Overall, we make use of 14 language extensions to make the proposed approach more accessible. The same mechanisms may not be available in other languages that otherwise support higher-kinded polymorphism or dynamically replacing a value by a sum type, meaning that logic programming will still be possible, but the notation may be more cumbersome.

8 Discussion

Our work shows how to introduce typed logic programming in a functional language operating on arbitrary types. Converting Prolog programs into Haskell has been a relatively straightforward exercise, although there are some considerations, which we discuss below.

8.1 Variable Scope

The scope of variables in our solution is, by default, global. A variable with a fixed name being used inside a function will be considered to be the same as a variable with the same name (and type) used elsewhere.

Consider the following Prolog example:

```
p(X) :- q(3).
q(X) :- X is 3.
```

Predicate `p(X)` holds for any value of `X`, because the variable is universally quantified locally in the definition of `p`. In contrast, if we write the Haskell predicate:

```
member x l = l =:= Cons x (V "tail")
          @| l =:= Cons (V "head") (V "tail")
             @@ member x (V "tail")
```

the function member will only hold if `x` is the first member of the list. All instantiations of `V "head"` refer *globally* to the same term, and so do all instantiations of `V "tail"`. Because the latter is not possible for finite lists, `member` will only hold if the first equation holds.

To work around this limitation, the function `exists` creates free variables. While this works around the issue, it does introduce a small degree of complexity in the way that logic programming can be used.

8.2 Variable Names and Types

The constructor `V` for `Terms` produces terms of any type, but two mentions of the same variable name in the same scope have different types. In our current solution, a variable is uniquely identified by its name *and* its type. Hence, using the same variable name in multiple places may lead to unexpected results. For example, in the following code, GHC does not know that both "x"'s refer to the same list:

```
> repl $ V "x" =:= Cons 1 (Nil :: ListF Term Int)
      @@ V "x" =:= (Nil :: ListF Term (NatF Term))
```

An alternative manifestation of the same problem is that the compiler may consider a variable's type ambiguous even if the same name is used elsewhere. For example, in the following query, the compiler does not understand that both `V "x"` refer to the same element:

```
> repl $ isLength 1 (V "x")
      @@ member (1 :: NatTerm) (V "x")
```

The second `"x"` must be a list of natural numbers because it is used as argument to `member` and the type of the first argument is given, but the first variable `(V "x")` may be a list of any numeric type.

A workaround this problem is to introduce a Haskell variable `x = V "x"`, and then use that variable instead. In general, adding the types to variables helps the type checker even in cases when it would seem obvious to the reader that the types are not ambiguous.

8.3 Negation and Unification

In this paper we introduced an operator `(=/=)`, which is the negation of `(=:=)`, and fails if the two terms unify. More specifically, that predicate will fail if, at the time when the unification algorithm tries to unify the two terms, they can unify. This means that, for example, if one of them is a variable, then the unification will always succeed, and so `(=/=)` will always fail. This makes writing some predicates either not possible with our current implementation, or not convenient.

Note that his problem is known in logic programming. The Prolog predicate `\=/2`, sometimes written as `\==/2`, behaves in the same way as our `(=/=)`. Some implementations of Prolog introduce a separate function `dif/2` that delays evaluation and carries with it a constraint. We consider that future work.

8.4 Variable Binding and Multiple Rules

Prolog predicates are normally defined by multiple rules, where the head of the rule can pattern match on the arguments, and the scope of variables is only that rule. Pattern matching in Prolog only indicates that a given input can unify with the given pattern, but it says nothing about whether the call to that predicate passed a free or ground term in that position.

In contrast, Haskell pattern matching checks the current value of an input, not whether unification is possible. For that reason, we have to capture variables without applying pattern matching, and use `(=:=)` in definitions to split the inputs into their component parts.

It is also worth noting that languages like Prolog specify multiple rules in separate definitions, while we have had to specify multiple rules within each definition.

Some extensions in the Haskell compiler GHC could make this process easier on the user, although, at the time of writing, we were not able to use such extensions to make pattern matching more convenient or more similar to Haskell or Prolog pattern matching. Making such a process convenient may require the development of new GHC extensions.

8.5 Defining Extensible Types

As currently proposed, users have to write an extensible type counterpart of the types they want to work with, and turn the values of the latter type into `Terms`, and ground terms back into values of the original, non-extensible type. Although we have tried to simplify that process with generic programming aids, there is some overhead to using our solution. The use of a library like `barbies` (Barbies, [n. d.]), or some form of template meta-programming, could help generate the extensible type that matches a given algebraic datatype. We consider that future work.

8.6 Anonymous Variables

Prolog allows the user of an underscore in place of a term, to indicate that the value in place should unify, but not capture it. For example, one can define a unary predicate `p` that holds for any value, or use anonymous variables in the right hand side of a rule, or in a query.

```
p(_).
r(X) :- q(_,X).
```

This is equivalent to introducing a free variable for each place where an underscore is used.

Our implementation supports the first kind of underscore simply because Haskell's wildcard pattern (also an underscore), works in the same way. Users of our implementation can, for example, define:

```
p :: NatTerm -> Predicate
p _ = succeed
```

Supporting the second kind of anonymous variable remains as future work.

9 Related Work

Functional-Logic Programming Languages The creation of languages that integrate logic and functional programming using theoretical frameworks and efficient implementations has been subject to prior study (Bellia and Levi, 1986; DeGroot and Lindstrom, 1986). Languages in this category include Babel (Moreno-Navarro and Rodríguez-Artalejo, 1988), K-LEAF (Giovannetti et al., 1991), ALF (Hanus, 1991), Curry (Hanus et al., 1995), and Escher (Lloyd, 1999), which support a functional style, and Gödel (Hill and Lloyd, 1994), Mercury (Somogyi et al., 1996), and λProlog (Nadathur and Miller, 1986), which embrace a logic programming style. Curry, in particular, is strongly inspired by Haskell but incorporates logical variables and uses narrowing as operational semantics to compute the value of expressions with free variables (Hanus, 2013). Instead of creating a new language, our work shows how an existing functional language can be empowered with logic programming capabilities, without additional compiler extensions and without calling an external logic programming engine.

"Functions" in Logic Programming Languages Prolog includes limited higher-order capabilities like `call/N` and `apply/3` (Naish, 1996). The Prolog implementation Ciao (Hermenegildo et al., 2012) allows, via its metaprogramming libraries, using predicates in a functional style, treating the last argument as the result of the function. In our case, functions and higher-order come built-in with the host language and are immediately exploitable by programmers. Furthermore, because we rely on a statically typed language, our approach provides a level of static safety. Prolog has previously been extended with static types (Barbosa et al., 2022; Mycroft and O'Keefe, 1984; Schrijvers et al., 2008), but these extensions are not integrated in the most widely used Prolog systems.

Logic Programming Embeddings Prior attempts at embedding logic programming in functional languages by Spivey and Seres (1999), Claessen and Ljunglöf (2001), Solanki (2012) and Elliott and Pfenning (1991) require adapting the types by hand to use them in logical predicates. Work by Kosarev and Boulytchev (2016) to embed relations in O'Caml require introducing explicit calls to *projections* to go from terms to functional values, and *injections* to go from functional values to terms. In contrast, our approach is applicable to arbitrary algebraic data types, and we provide generic programming aids to facilitate operating with them, significantly simplifying the process and broadening the applicability of our proposal. The application of a systematic extension pattern, rather than hand-coded extensions, leads to more regular and predictable ways to add variables to any algebraic data type.

Another difference between these embeddings and our work is in how predicate evaluation is implemented. Prior embeddings use an interpreter based on an evaluation monad (Claessen and Ljunglöf, 2001; Schrijvers et al., 2009; Solanki, 2012; Spivey and Seres, 1999), make predicates data *streams* and explicitly introduce a backtracking lazy stream monad in an otherwise strict setting (Kosarev and Boulytchev, 2016), or use continuation passing style and exceptions to evaluate logic programs more efficiently (Elliott and Pfenning, 1991). In contrast, we use an internal representation of predicates as a tree of unification constraints, parameterized by a counter that can be used to generate free variables. Our encoding provides a more fine-grained control of the counter than what we could obtain by directly using a (lazy) state monad, which introduced an additional degree of laziness that was crucial to generate solutions efficiently. The tree is later traversed to perform unification, rendering a simplified tree, which is flattened to a list to explore the solutions and present them to users in the REPL. Another difference in implementation between miniKanren (Kosarev and Boulytchev, 2016) and our work is that the former uses a type-unsound internal representation encapsulated behind a type-safe API. Our approach maintains type safety by avoiding the use of unsafe language features for implementing unification, and term unification can only be applied to terms of the same type.

These embeddings also differ from our work in terms of expressiveness: for example, we only support equality constraints, and implement a limited form of disequality using a (weak) form of negation via cuts, whereas miniKanren and the work of Schrijvers et al. (2009) support both equality and disequality.

Higher-kinded Type Parametrizations Our proposed solution uses a specific technique to parameterize types with higher kinds. Alternative techniques have been proposed, and are being used in compilers and other tools. Najd and Jones (2017) proposed extending trees by adding a parameter `f` to every branch of an algebraic data type's definition. Najd's approach does not apply the parameter `f` to elements inside the type, and uses type families to customize the behavior for each constructor. The work of Najd et al. also allows introducing `Term`-like wrappers around all elements of the abstract data type. Nevertheless, we find extensible types to be straightforward due to the minimal work required to let *any* part of a datatype be replaced with a variable by merely applying `Term` to the extensible type.

A number of existing tools and libraries generate extensible types and faciliates working with them. BNFC (Forsberg and Ranta, 2004) uses an approach similar to extensible types to generate, from a given grammar, an Abstract Syntax Tree that is parameterized over an argument `f`. The library `barbies` (Barbies, [n. d.]) implements a collection of generic mechanisms to work with types parameterized by a functor applied everywhere in a type definition, akin to extensible types. The libraries Barbies-th (Barbies-th, [n. d.]) or Higgledy (Higgledy, [n. d.]) could help generate higher-kinded variants of type definitions in Haskell. We have yet to investigate how our solution could leverage the support for extensible types provided by the aforementioned libraries and tools.

10 Future Work

This paper has shown how to embed logic programming in a statically typed functional programming language. To that end, we used extensible types to replace any portion of an algebraic datatype with variables, and provided a mechanism to express unification constraints between values with variables. We further extended the language with boolean connectives, and ways to introduce free variables. We showed by example that we can leverage the host language's type inference and higher-order to make code reusable without sacrificing type safety. We closed our discussion with an overview of our implementation, a discussion of limitations, and an evaluation of the differences with other approaches.

The examples presented in this paper show how to evaluate predicates in a REPL. In future work, we will show how to use the logic programming facilities described in this paper to write functional programs.

The same approach proposed in this paper could be used to represent other kinds of constraints. This would allow us to implement constraint-logic programming in Haskell. Explicitly introduction disequality constraints in our language would overcome the limitations of our current implementation of disequality using "negation as failure" (akin to Prolog's).

We are currently exploring how the language provided in this paper could be used to generate values more optimally in property-based testing, since constraints could be carried with types, rather than inefficiently generating values and then filtering based on constraints (thus discarding many values).

In this paper we have not discussed benchmarking our unification algorithm against an existing implementation, as it is a topic that deserves careful and detailed evaluation. We plan to conduct thorough benchmarks against known implementations, and to use them to identify bottlenecks that we could address.

Our experiments indicate that the semantics of the predicate evaluation functions we have implemented coincides with that of Prolog. In future work, we would like to carry out a more detailed and formal evaluation to compare our inference engine with Prolog's.

References

Barbies [n. d.]. Barbies. `https://hackage.haskell.org/package/barbies`.

Barbies-th [n. d.]. barbies-th. `https://hackage.haskell.org/package/barbies-th`.

João Barbosa, Mário Florido, and Vítor Santos Costa. 2022. Data Type Inference for Logic Programming. In *Logic-Based Program Synthesis and Transformation*, Emanuele De Angelis and Wim Vanhoof (Eds.). Springer International Publishing, Cham, 16–37.

Marco Bellia and Giorgio Levi. 1986. The relation between logic and functional languages: a survey. *The Journal of Logic Programming* 3, 3 (1986), 217–236. `https://doi.org/10.1016/0743-1066(86)90014-2`

Koen Claessen and Peter Ljunglöf. 2001. Typed Logical Variables in Haskell. In *2000 ACM SIGPLAN Haskell Workshop (Satellite Event of PLI 2000).* `https://doi.org/10.1016/S1571-0661(05)80544-4`

Doug DeGroot and Gary Lindstrom (Eds.). 1986. *Logic Programming: Functions, Relations, and Equations.* Prentice-Hall.

Conal Elliott and Frank Pfenning. 1991. A semi-functional implementation of a higher-order logic programming language. *Topics in Advanced Language Implementation* (1991), 289–325.

Markus Forsberg and Aarne Ranta. 2004. BNF converter. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell.* 94–95.

Elio Giovannetti, Giorgio Levi, Corrado Moiso, and Catuscia Palamidessi. 1991. Kernel-LEAF: A logic plus functional language. *J. Comput. System Sci.* 42, 2 (1991), 139–185. `https://doi.org/10.1016/0022-0000(91)90009-T`

Michael Hanus. 1991. The ALF system: An efficient implementation of a functional logic language. In *Processing Declarative Knowledge*, Harold Boley and Michael M. Richter (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 414–416.

M. Hanus. 2013. Functional Logic Programming: From Theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*. Springer LNCS 7797, 123–168.

M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. 1995. Curry: A Truly Functional Logic Language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*. 95–107.

M. V. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J. F. Morales, and G. Puebla. 2012. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming* 12, 1–2 (2012), 219–252. `https://doi.org/10.1017/S1471068411000457`

Higgledy [n. d.]. Higgledy. `https://hackage.haskell.org/package/higgledy`.

P. Hill and J.W. Lloyd. 1994. *The Gödel Programming Language*. MIT Press. `https://books.google.es/books?id=AYYWhZxwmw4C`

Dmitry Kosarev and Dmitry Boulytchev. 2016. Typed Embedding of a Relational Language in OCaml. In *Proceedings ML Family Workshop / OCaml Users and Developers workshops, ML/OCAML 2016, Nara, Japan, September 22-23, 2016 (EPTCS, Vol. 285)*, Kenichi Asai and Mark R. Shinwell (Eds.). 1–22. `https://doi.org/10.4204/EPTCS.285.1`

John W Lloyd. 1999. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming* 3, 1-49 (1999), 68–69.

José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. 2010. A generic deriving mechanism for Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell* (Baltimore, Maryland, USA) *(Haskell '10)*. Association for Computing Machinery, New York, NY, USA, 37–48. `https://doi.org/10.1145/1863523.1863529`

Juan José Moreno-Navarro and Mario Rodríguez-Artalejo. 1988. BABEL: A Functional and Logic Programming Language based on Constructor Discipline and Narrowing. In *International Conference on Algebraic and Logic Programming*. `https://api.semanticscholar.org/CorpusID:38448321`

Alan Mycroft and Richard A. O'Keefe. 1984. A polymorphic type system for prolog. *Artificial Intelligence* 23, 3 (1984), 295–307. `https://doi.org/10.1016/0004-3702(84)90017-1`

Gopalan Nadathur and Dale A. Miller. 1986. Higher-order logic programming. In *Third International Conference on Logic Programming*, Ehud Shapiro (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 448–462.

Lee Naish. 1996. Higher-order logic programming in Prolog. In *Proc. Workshop on Multi-Paradigm Logic Programming, JICSLP*, Vol. 96. 1–23.

Shayan Najd and Simon Peyton Jones. 2017. Trees that grow. *Journal of Universal Computer Science* 21, 1 (2017), 42–62.

Ivan Perez. 2023. Types that Change: The Extensible Type Design Pattern. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Functional*

Software Architecture (Seattle, WA, USA) *(FUNARCH 2023)*. Association for Computing Machinery, New York, NY, USA, 49–62. `https://doi.org/10.1145/3609025.3609475`

Tom Schrijvers, Vítor Santos Costa, Jan Wielemaker, and Bart Demoen. 2008. Towards Typed Prolog. In *Proceedings of the 24th International Conference on Logic Programming* (Udine, Italy) *(ICLP '08)*. Springer-Verlag, Berlin, Heidelberg, 693–697. `https://doi.org/10.1007/978-3-540-89982-2_59`

Tom Schrijvers, Peter Stuckey, and Philip Wadler. 2009. Monadic constraint programming. *Journal of Functional Programming* 19, 6 (2009), 663–697. `https://doi.org/10.1017/S0956796809990086`

Mehul Chandrakant Solanki. 2012. *Embedding Programming Languages: Prolog in Haskell*. Master's thesis. University of Northern British Columbia.

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. `https://doi.org/10.1016/S0743-1066(96)00068-4` High-Performance Implementations of Logic Programming Systems.

J Michael Spivey and Silvija Seres. 1999. Embedding prolog in haskell. In *Proceedings of Haskell*, Vol. 99. 1999–28.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704–0188*

1. REPORT DATE *(DD-MM-YYYY)* 01-08-2024	**2. REPORT TYPE** Technical Memorandum	**3. DATES COVERED** *(From - To)*

4. TITLE AND SUBTITLE Logic Programming with Extensible Types	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Ivan Perez, Angel Herranz	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Ames Research Center, Moffett Field, CA 94035	**8. PERFORMING ORGANIZATION REPORT NUMBER** L–XXXXX
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	**10. SPONSOR/MONITOR'S ACRONYM(S)** NASA
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) NASA/TM–20240010266

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 59
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES
An electronic version can be found at http://ntrs.nasa.gov.

14. ABSTRACT
Logic programming allows structuring code in terms of predicates or relations, rather than functions. Although logic programming languages present advantages in terms of declarativeness and conciseness, the introduction of static types has not become part of most popular logic programming languages, increasing the difficulty of testing and debugging of logic programming code. This paper demonstrates how to implement logic programming in Haskell, thus empowering logic programs with types, and functional programs with relations or predicates. We do so by combining three ideas. First, we use extensible types to generalize a type by a parameter type function. Second, we use a sum type as an argument to introduce optional variables in extensible types. Third, we implement a unification algorithm capable of working with any data structure, provided that certain operations are implemented for the given type. We demonstrate our proposal via a series of increasingly complex examples inspired by educational texts in logic programming, and leverage the host language's features to make new notation convenient for users, showing that the proposed approach is not just technically possible but also practical.

15. SUBJECT TERMS
logic programming, functional programming, domain-specific languages, type-level programming

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Information Desk (`help@sti.nasa.gov`)
U	U	U	UU	36	**19b. TELEPHONE NUMBER** ***(Include area code)*** (757) 864-9658

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18