klogic: miniKanren in Kotlin

YURY KAMENEV, No affiliation, Russia
DMITRII KOSAREV, St. Petersburg State University, Russia
DMITRY IVANOV, No affiliation, Russia
DENIS FOKIN, No affiliation, Russia
DMITRII BOULYTCHEV, St. Petersburg State University, Russia

One of the distinguishable features of embedded domain-specific languages, like miniKanren, is an easy integration with a host language. Recently, we were asked to port our relational programs in OCaml to Java Virtual Machine (JVM). As a result, we got a miniKanren implementation in Kotlin, which resembles OCanren (miniKanren in OCaml). In this paper, we describe the peculiarities of a relational language implementing on JVM using Kotlin.

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

Additional Key Words and Phrases: miniKanren implementation, Kotlin, relational programming

1 INTRODUCTION

One of the appealing features of relational programming is an easy interaction between general-purpose and relational programming languages. We often find it useful to prototype a solution in miniKanren, and later improve the performance by splitting big relational programs into a number of relational programs, connected by a functional glue. We usually use OCaml and the OCanren DSL to do this, but recently we were asked to make our work executable on JVM.

The straightforward solution to execute OCanren in JVM would be using an OCamlJava [4] compiler, but this project looks quite dead. Using JNI could be error-prone (for inexperienced JVM users like us). Another approach would be a source-to-source transformation from OCaml to a JVM-friendly language. We left it as a future work, because, at first, it is difficult for us to estimate the difficulty of the task, and, second, it requires a working implementation of a relational DSL in JVM.

There are plenty of relational libraries for JVM languages, e.g. for Scala (Scalogno [1]) and Core.Logic. However, according to TIOBE [2], Kotlin is the most popular JVM language except for Java itself, but Java lacks the syntax needed for a compact miniKanren implementation. In our mind, more professional developers and enthusiasts can give into a new Kotlin library and contribute to it. Our customer, who is interested in relational programming in OCanren as well values Kotlin very much.

As a result, we developed a relational programming library klogic[1] in Kotlin. The reference implementation was OCanren, but we needed to do a few things differently because of JVM quirks. In this paper we report the main technical decisions about Kotlin implementation, recollect the peculiarities of the OCanren implementation in OCaml, discuss the programming experience from an OCaml developer point of view, and do some benchmarking.

2 OCANREN REMINDER

The klogic implementation is based on OCanren: typed embedding of miniKanren to OCaml. In this section, we describe basic blocks of OCanren, their implementation in klogic is given in the next section. A few features (for example, disequality constraints) in klogic and OCanren are implemented similarly and will be left out of discussion.

[1]https://github.com/UnitTestBot/klogic (access date: 2023-06-15)

Unification is being performed over a tree of pointers, which represent OCaml values in the runtime. Most algebraic values are the nodes of the tree, basic types and algebraic constructors without arguments are the leaves. Inspection and reconstruction of the trees is performed via a low-level non type-safe interface.

No proper OCaml types could not be assigned to values injected into logic domain, because their representation differs. This is done for performance considerations.

Types for logic representations. In typed relational programs you can find three sorts of types: for non-relational ground representations, logic domain, and types of relational search results, that represent values *reified* (extracted) *from logic domain.* For the reified values we use a straightforward solution: we declare the algebraic data type to represent either logic variables or values.

```
type 'a logic = Var of var_idx | Value of 'a
```

In principle, we can use the same `logic` type for representations in a logic domain. It will lead to poor performance, which is demonstrated in [6]. These days OCanren uses slightly different types for a logic domain than the ones described in [6], and we add details about it. For every subvalue embedded into the logic domain, we decorate a type of this subvalue with the predefined type `type 'a ilogic`. For example, integer constants will have in the logic domain type `int ilogic`. The same holds for other types considered primitive in OCanren: strings, integers, floats, booleans. In Scheme context, the closest analogue would be symbols. The injection of primitive values a to logic domain could be done using predefined function `inj`.

```
val inj: 'a -> 'a ilogic
```

The injection of user-defined algebraic data types is more complicated. We want to allow the placement of logic values in all subparts of data type, not only in positions of type variables. To achieve that, we abstract away all concrete type parameters, and get "fully abstract data type". We can parameterize it by substituted types to get the type isomorphic to the original one, or parameterize with `ilogic` types to get the type for logic domain, or parameterize with the type `logic`, to get a type of reified representation. In Listing 2 one can see three types: the one without logic variables, the logic counterpart, and for reified results. One can note how the second type is being constructed by addition of `ilogic` type everywhere, and the reified type by replacing `ilogic` with `logic`.

```
(int * bool, 'a) fa_list as 'a
((int ilogic * bool ilogic) ilogic, 'a) fa_list ilogic as 'a
((int logic * bool logic) logic, 'a) fa_list logic as 'a
```

The injection of user-defined types is essentially an application of a constructor to injected values, followed by the primitive injection `inj`. In the previous implementation [6], the representation of injected types had two type parameters to track ground and reified types of values. This approach requires predefined OCaml functors, one functor per type arity. In the current OCanren implementation, this is not needed, which allows to get rid of functors and control reified type with more flexibility.

Reification. Reification for user data type is a composition of a primitive predefined reifier (for strings, integers, floats, and booleans), a predicate to distinguish variables from values, and a fixpoint combinator to get reifiers for recursive values. All reifiers are two-parametric types which track in the parameters the type in the logic domain, and the type we are going to reify into. In practice, we found it convenient to have two kind of reifiers. The default one reifies to "logic" types, which can represent variables using `logic` type mentioned above. It could be considered a general form of reification. Another one is called *projection*, which reifies from a logic domain to an original

```
type 'a list = [] | (::) of 'a * 'a list
type ('a, 'b) fa_list = Nil | Cons of 'a * 'b
type 'a iso_list = ('a, 'a iso_list) fa_list
type 'a injected_list = ('a, 'a injected_list) fa_list ilogic
type 'a logic_list = ('a, 'a logic_list) fa_list logic
```

Fig. 1. A few examples of types in OCanren using list data type. The list type is a default linked list defined in OCaml, conventional constructors have a special treatment in the parser. The fa_list is a fully abstract version of a list with constructors renamed. The iso_list is a definition via fa_list of the type isomorphic to the original list. We can add ilogic / logic types to the definition of the type iso_list to get the types for logic/reified domains.

ground representation, or raises an exception when a free variable is encountered. It reifies to the type without holes for logic variables, and these sort of types is more approachable for integration of functional and relational code. The projection should be used for relations, when we are sure that the answer of relational search is ground.

```
val reify_list : ('a, 'b) Reifier.t ->
  ('a injected_list, 'b logic_list) Reifier.t
val prj_exn_list : ('a, 'b) Reifier.t ->
  ('a injected_list, 'b list) Reifier.t
```

In [6] we track the type of reified values during injection process, and the type of reified values is fixed. Now we outsource to the user the construction of reified values, and the one is empowered to construct values of a desired type. It is possible to project a list from the logic domain to the standard OCaml list type. In the previous approach, we were limited to reification to `'a iso_list`. It required a manual conversion to default OCaml lists, which was cumbersome.

The klogic implementation will reuse these ideas, because both OCanren and klogic are embedded to a typed language. There is no two types of reifiers per every data type in Scheme, a single implicit reifier is enough.

Primitives. In original miniKanren for Scheme, most of the primitives are implemented using a macro system. In OCanren, we apply similar macro for `fresh`, but conde is just a function that takes a list of goals. We also have infix binary high-order relations for disjunction and conjunction.

3 IMPLEMENTATION

In this section, we describe peculiarities of unification with user-defined types in klogic.

3.1 Types for logic representation

In the klogic, we introduce a special interface called $Term$ to represent the logic domain. There are two types of logic terms — logic variables and logic values (that can store other logic terms inside). Logic variables are represented as inline wrappers of integer index which distinguishes variables; logic values can store any value, logic or not. So, for a simple implementation (that an original Scheme[2]) it would be enough to represent the logic domain by declaring the interface $Term$ with one existing inheritor — the inline class Var — and allowing users to inherit the $Term$ for implementing user’s logic types.

[2]https://github.com/michaelballantyne/faster-minikanren (access date: 2023-06-06)

Unfortunately, this implementation would have an important disadvantage — it allows users to write goals that unify values of different types (for example, natural numbers and linked lists). These sorts of unification should fail, and it sounds reasonable in a typed language to forbid such unifications at compile time. In general, it means that each logic variable has to be created with some known logic type to which it can be reified and with terms of which it can be unified. To handle such a case, we decided to use parameterized types that are represented by Kotlin generics.

Actually, we have three possible types of unifications:

- unification of two logic values, both of the same type;
- unification of two logic variables, both over the same type;
- unification of a logic value and a logic variable of the same type.

These observations motivate us to implement unification function `unify` with the following signature.

```
fun <T> unify(first: Term<T>, second: Term<T>) = ...
```

But this approach has a drawback — it allows the creation of logic variables over non-logic types because the parameter does not have any restrictions. The key solution to handle it is making the parameter a logic type too. It leads to the following self-recursive declarations.

```
interface Term<T : Term<T>> { ... }
class Var<T : Term<T>> : Term<T> { ... }
```

Having these declarations, we introduce type-checking at compile time for unifications, but another problem arises — implementing a mechanism of unifications for arbitrary logic types. A unification requires traversing all fields of a logic value that need to be unified with an another logic value. It can be done using Java reflection, but such an approach has significant shortcomings. Firstly, using reflection leads to a substantial decrease in performance. Secondly, with reflection, we lose flexibility because we cannot allow a user to specify what fields should be unified and what should not. We came up with another solution. An abstract implementation of unification is provided in the base class for logic terms. It walks both logic terms and unifies all values that are defined for current logic term by a user (Listing 1). As a result, to define a new logic term a user should inherit from the `CustomTerm` and implement two abstract properties — `subtreesToUnify` and (optionally) `subtreesToWalk`.

Listing 1. Declaration of user-defined type CustomTerm in KLOGIC (a sketch)

```kotlin
sealed interface Term<T : Term<T>> {
    fun unify(
            other: Term<T>,
            unificationState: UnificationState
    ): UnificationState? {
        val walkedThis = walk(unificationState.substitution)
        val walkedOther =
            other.walk(unificationState.substitution)

        return walkedThis.unifyImpl(walkedOther, unificationState)
    }

    fun unifyImpl(walkedOther: Term<T>,
        unificationState: UnificationState
    ): UnificationState?
}

@JvmInline
value class Var<T : Term<T>>(val index: Int) : Term<T> {
    override fun unifyImpl(
        walkedOther: Term<T>,
        unificationState: UnificationState
    ): UnificationState? {
        ...
    }
}

interface CustomTerm<T : CustomTerm<T>> : Term<T> {
    val subtreesToUnify: Array<*>

    val subtreesToWalk: Array<*>
        get() = subtreesToUnify

    override fun unifyImpl(
        walkedOther: Term<T>,
        unificationState: UnificationState
    ): UnificationState? {
        ...
    }
}
```

3.2 Variables construction

Each logic variable is identified by its unique integer identifier, so to be able to create fresh variables we need to maintain a mechanism that creates new unique identifiers. We implemented it quite simply by storing an integer index of the last created fresh variable and issuing the incremented value of this index for a new fresh variable.

In addition to the variable index, among unifications, we need to maintain a set of current added constraints (for now we have only disequality constraints) and substitution of created logic variables to other logic terms (values or variables). We reused the standard concept of a *state* (Fig. 2) — an immutable union of the constraints and the substitution (represented by a persistent map from logic variables to logic terms), which changes by unifications.

Listing 2. Definition of the state in KLOGIC

```
data class State(
    val substitution: Substitution,
    val constraints: PersistentSet<Constraint<*>> =
        persistentHashSetOf(),
)
```

Speaking about creating fresh variables, there is an important detail in implementation. The creation introduces a *lazy* goal, i.e. inserts a *delay*, the user-accessible interface should encourage users to create with a single delay a few fresh variables at once. So, we need to have many different functions for creating different numbers of fresh variables. In some programming languages (most of those are descendants of C programming language, for example), we can automatically generate such functions at the compile-time using *macro* mechanism. For some reason, KOTLIN in particular, and JAVA in general, do not have macros, although some of their features may be replaced by processing of annotations. Moreover, the number of method's parameters in JAVA is limited to 255[3], and creating more than 255 fresh variables at once is impossible. So, in theory different functions for creating different numbers of fresh variables could be generated somehow using Java annotations. For now, we have a few predefined implementations (from 1 to 8 fresh variables, to be exact), and functions to create more than 8 fresh variables at once has to be implemented manually by a user.

The absence of macro mechanism in the language affects the way we write relational programs. All MINIKANREN primitives should be defined as functions, which are evaluated in call-by-value strategy. In some cases, it requires an explicit inverse-eta-delay insertion before a recursive call of relation. KLOGIC and OCANREN have primitives for that, but in SCHEME it is not needed because `conde` and `fresh` primitives are implemented using a macro. We modified the code of our benchmarks to make execution traces the same as in SCHEME. As a result, sometimes relation definitions are not an idiomatic KLOGIC/OCANREN code.

There is an important detail in implementing and using functions for creating fresh variables connected with KOTLIN type system. Since that language was not designed to have type inference, and the usage of overloaded functions is widespread, we are obliged to manually specify types of created fresh variables when using `fresh` — consider an example in Fig. 2. On the same listing, there is a subtlety about unification of empty logic lists. The first `` `===` `` unifies a logic variable with an empty logic list. A swap of these two arguments is allowed in SCHEME and OCANREN but not in KOTLIN: type inference can't guess the type of elements of an empty logic list using the

[3]https://docs.oracle.com/javase/specs/jvms/se20/html/jvms-4.html#jvms-4.3.3 (access date: 2023-06-06)

```
fun <T : Term<T>>
    appendo(x: ListTerm<T>, y: ListTerm<T>, xy: ListTerm<T>): Goal
  =
    ((x `===` nilLogicList()) `&&&` (y `===` xy))
  `|||`
    freshTypedVars<T, LogicList<T>, LogicList<T>>
        { head, tail, rest ->
            (x `===` head + tail) `&&&`
            (xy `===` head + rest) `&&&`
            appendo(tail, y, rest)
        }
```

Fig. 2. Appendo in KLOGIC. Infix binary operators for conjunction, disjunction, and unification should be written in backticks. Overloaded primitive for creation of fresh variables (almost always) requires types specification.

type of a logic variable. In the benchmark implementations, we adapted many unifications to make unification traces of SCHEME, OCANREN and KLOGIC exactly the same.

4 KLOGIC VS. OCANREN

In this section, we discuss programming with KLOGIC from OCANREN programmer's point of view.

The first thing that catches the eye: type annotations are everywhere. Sometimes we can omit type annotations for fresh variables, for example, when a variable is not used, but these cases are rare. This aspect of KLOGIC is not a particular implementation decision, but rather an artifact of KOTLIN. The host language is not designed to have a powerful type inference, and in the presence of overloaded functions (for example, the infix function + from Listing 2 which is a relational version of `cons`) the compiler's ability to infer types is limited.

The `freshTypedVars` primitive could be annotated by types in two ways: either in angle brackets or without angle brackets but near every introduced variable. For example, you could create fresh logic variables either via `freshTypedVars<LogicInt, LogiInt> { n, m -> ... }` or `freshTypedVars { n: Term<LogicInt>, m: Term<LogicInt> -> ... }`. With the first approach the types and the names of variables are textually separated and readability is reduced. In the second approach we need to specify explicitly, that we create a variable of a type of logic values (just `LogicInt` would be a ground integer), and type annotations become longer. Sadly, the ex-OCANREN developer should decide between two suboptimal approaches. We speculate that ex-SCHEME developer would feel even more frustration.

The proposed encoding of user-defined types is a decent step forward, in comparison to another KOTLIN implementation[4] we found. The KOTLINKANREN implementation proposed to use universal representation for values in logic domain, and user data types should be converted to this representation and back, which could be error-prone and could raise issues related to user-defined types' representability. The KLOGIC approach is better, but in comparison to OCANREN we could wish that the approach would require less boilerplate code. Ideally, the method `subtreesToUnify` should be generated using some macro/annotation mechanism to protect developers from mistakes.

[4]https://github.com/neilgall/KotlinKanren (access date: 2023-06-06)

OCanren doesn't allow us to skip non-logic subvalues of a logic value during unification currently, and klogic is theoretically better in this approach. Sadly, we don't have a good example to demonstrate the usefulness of this feature.

Klogic currently lacks a few optimizations present in OCanren and faster-miniKanren. The implementation of disequality constraints is very straightforward. Also, we don't optimize the unification with recently created fresh variables (also known as *set-var-val* optimization in faster-miniKanren).

5 BENCHMARKS

We compare faster-miniKanren in Racket with OCanren in OCaml and klogic in Kotlin in a few performance tests involving relational arithmetic [5] and Scheme relational interpreter [3]. We picked the same relations to benchmark as the paper [6] did. All benchmarking was performed on the desktop machine Intel© Core™ i7-4790K (16Gb RAM). Some software was taken from the official Ubuntu 23.04 x86_64 repository: Racket 8.7 (compiled to native code using Chez backend) and OpenJDK 17.0.7 (as the last Java LTS version). OCaml compiler 4.14+flambda was installed using OPAM[5] package manager. Running benchmarks was implemented using language-specific libraries: benchmark[6] for OCaml, benchmark[7] for Racket and JMH[8] for Kotlin.

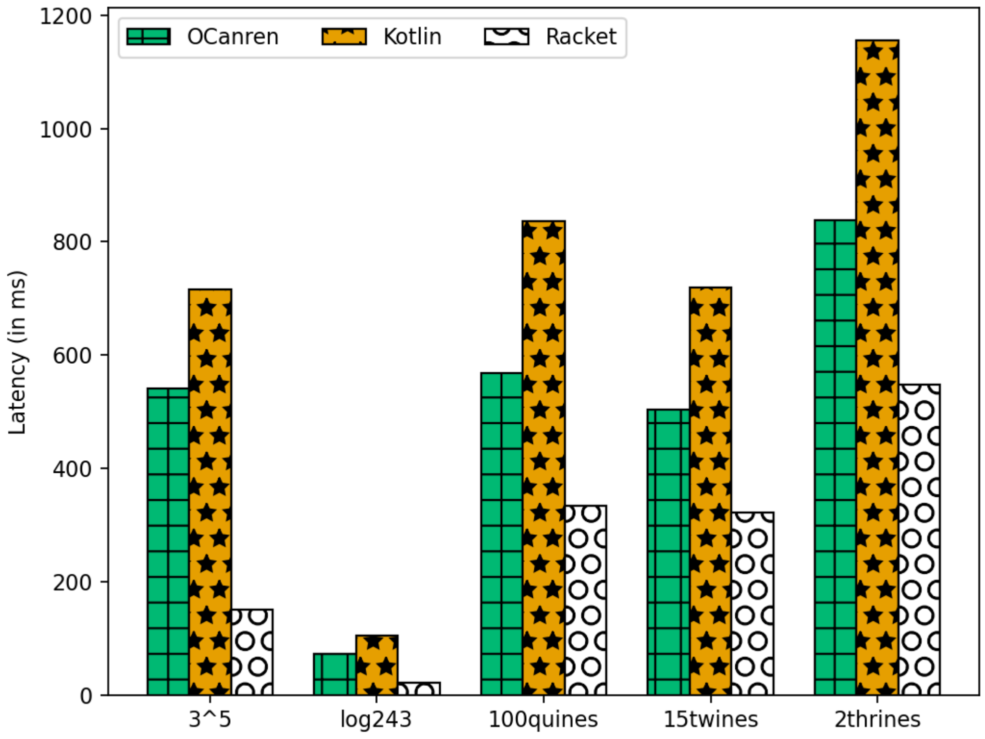
In the accompanying repository[9] we can find unification counts and unification traces for relations being benchmarked. It also has a submodule for a modified klogic implementation where traces of unifications can be toggled on/off using an environment variable. Unfortunately, this check is not optimized out by just-in-time compilation in JVM, and we need to comment it out manually. That's why the benchmarks implementation in klogic lives in another repository[10].

The table 3 demonstrates that if we worry only about performance, the faster-miniKanren implementation should be recommended instead the other implementations. Today klogic has a naive implementation of disequality constraints, which decreases performance of relational interpreter. However, disequality constraints are not involved to the Oleg numbers benchmarks. We can't explain why klogic unperformed, but the set-var-val optimization from faster-miniKanren should be applicable there. In the draft of the paper, we presented more auspicious benchmarks of klogic but the search order had not been the same between three implementations. The big changes of numbers demonstrates that evaluation of the performance as number of unifications per second could be misleading. The context switching of relational streams may seriously affect performance (it has been observed already [7]).

6 CONCLUSIONS AND FUTURE WORK

We presented klogic — a library for relational programming in Kotlin. It is designed to have a typed representation of logic values, which prevents developers from a certain class of mistakes. The disequality constraints are available, but other domain-specific constraints aren't (we believe that *absento* is not required for our representation of logic values). The implementation is rather straightforward in a few places. For example, optimizations from faster-miniKanren (lazy disequality constraints and storage for variable bindings inside variables) are not yet implemented. At the moment performance is slightly worse comparatively to OCanren, and we speculate that missing optimizations may change that.

[5]https://opam.ocaml.org/ (access date: 2023-08-22)
[6]https://github.com/Chris00/ocaml-benchmark (access date: 2023-08-22)
[7]https://github.com/stamourv/racket-benchmark (access date: 2023-08-22)
[8]https://github.com/openjdk/jmh (access date: 2023-08-22)
[9]https://github.com/Kakadu/miniKanren_exec_order/tree/tracing (access date: 2023-08-22)
[10]https://github.com/Kakadu/klogic/tree/tracing (access date: 2023-08-22)



	Unification count	OCanren (in ms)	Kotlin (in ms)	Racket (in ms)
3^5	433854	541.80	715.06	150.63
$log_3 243$	56277	72.48	105.08	22.24
100 quines	150732	567.75	836.67	334.36
15 twines	148525	502.90	719.67	321.27
2 thrines	224658	838.77	1155.47	548.38

Fig. 3. Performance results for various relations (in OCanren, Kotlin and Racket) involving Oleg numbers [5] (time of first answers of exponentiation 3^5 and inverse) and Scheme relational interpreter [3] (100 first quines, 15 twines and 2 thrines). The search order in the implementations is the same. On the Y axis we have latency in milliseconds (less is better).

The process of making execution traces for tree implementations exactly the same took much more time than we initially expected. In future, it would be great to have an automatic translator of relational programs between klogic, OCanren, and Racket, because the manual comparison of implementations' code is cumbersome.

REFERENCES

[1] Nada Amin, William E. Byrd, and Tiark Rompf. 2019. Lightweight Functional Logic Meta-Programming. In *Programming Languages and Systems*, Anthony Widjaja Lin (Ed.). Springer International Publishing, Cham, 225–243.
[2] TIOBE Software BV. 2022. *TIOBE Index.* TIOBE Software BV. Retrieved June 9, 2023 from https://www.tiobe.com/tiobe-index/
[3] 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)*. ACM, New York, NY, USA, 8–29. https://doi.org/10.1145/2661103.2661105

[4] Xavier Clerc. 2013. OCaml-Java: OCaml on the JVM. In *Trends in Functional Programming*, Hans-Wolfgang Loidl and Ricardo Peña (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 167–181.

[5] 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.

[6] Dmitry Kosarev and Dmitry Boulytchev. 2016. Typed Embedding of a Relational Language in OCaml. *Electronic Proceedings in Theoretical Computer Science* 285 (2016), 1–22. https://doi.org/10.4204/EPTCS.285.1

[7] Dmitry Rozplokhas and Dmitry Boulytchev. 2022. Scheduling Complexity of Interleaving Search. In *Functional and Logic Programming*, Michael Hanus and Atsushi Igarashi (Eds.). Springer International Publishing, Cham, 152–170.