MicroKanren in J: an Embedding of the Relational Paradigm in an Array Language with Rank-Polymorphic Unification

RAOUL SCHORER, Geneva University Hospitals, Switzerland

This work presents a reimplementation within the array programming paradigm of MicroKanren, a minimal relational language designed to be embedded in host languages as a library. The particularities of the implementation relative to the array paradigm are detailed. The implementation is discussed in relation to the reference Racket version and a rank-polymorphic unification algorithm is presented. Those elements are discussed in light of future prospects.

CCS Concepts: • **Computing methodologies** → *Logic programming and answer set programming*; • **Software and its engineering** → **Language types**.

Additional Key Words and Phrases: array programming, logic programming, microKanren, miniKanren, relational programming, streams, unification

ACM Reference Format:
Raoul Schorer. 2020. MicroKanren in J: an Embedding of the Relational Paradigm in an Array Language with Rank-Polymorphic Unification. 1, 1 (July 2020), 18 pages. https://doi.org/0000001.0000001

INTRODUCTION

Logic programming allows a high level of abstraction in both code and problem solving concepts. Language representatives of this paradigm of which Prolog is the most well-known traditionally aim at a declarative approach and are remarkably versatile [21]. MiniKanren and its even more minimalist sibling language MicroKanren offer an alternative and distinct tradeoffs with similar capabilities in the context of relational programming [6, 9]. MicroKanren semantics are designed to be easily transcribed to a variety of host languages, to the point that implementing a shallow embedding of Micro- or MiniKanren as a library has become a rite of passage for newcomers to the Kanren language family. The purely functional semantics of MicroKanren translates to a great variety of programming paradigms with little change in general syntax and operators, yielding a somewhat portable notation for relational programs [5]. It is also interesting that shallow embedding allows Micro/MiniKanren to inherit host language characteristics. A notable illustration of this principle can be found in OCanren, an embedding of Minikanren that cooperates with the underlying OCaml type system [20]. It is therefore worthwile to explore other well-chosen host languages and their interaction with the relational paradigm. This work presents an implementation of MicroKanren in the J array language and discusses the resulting features in relation to a reference implementation in Racket as described by Hemann *et al.* [12]. We begin by a short presentation of the array language J and the features that motivated its use. Next, we describe the main points of the implementation of MicroKanren on top of J. We then examine example programs and

Author's address: Raoul Schorer, Dept. of Acute Medicine, Geneva University Hospitals, Rue Gabrielle-Perret-Gentil 4, Geneva, 1205, Switzerland, raoul.schorer@hcuge.ch.

benchmarks. In the final section, we discuss related works as well as limitations and future prospects before concluding. The main contributions of this paper are:

(1) to show that array rank operators allow flexible unification over multidimensional arrays.
(2) to provide a prototype Kanren implementation based exclusively on arrays with the long term goals of
   - exploring adaptations of the logic programming paradigm to vector processing.
   - developing a logic language hosted on vector platforms such as graphical processor units (GPU) or that can make good use of single instruction multiple data (SIMD) instruction sets.

CHOOSING J

The array language J is a sibling of APL restricting its character set to ASCII. Interested readers unfamiliar with J may find introductions and tutorials on the website (jsoftware.com), which will enable them to understand the code below. Additionally, the J dictionary (jsoftware.com/wiki/nuvoc) contains accessible descriptions and examples applications of all language primitives. Array languages are characterized by a mostly-pure functional programming paradigm with severe syntactic constraints, and J is no exception. The power of array languages resides in their vast collection of primitives usually implemented in a low-level language allowing high-level array manipulation at speed. While J has most of the features that make a good Kanren host language (garbage collection, object orientation, metaprogramming through quotation), some syntactic features such as the strict limit of two arguments per function make the implementation less straightforward [10]. Below, we describe some features that were most helpful in the implementation of MicroKanren in J.

Rank Polymorphism. Rank is a crucial concept in J and underlies the contributions made by this paper. The rank of an array is defined as its number of dimensions. In an attempt at improved practicality, primitives were made rank-polymorphic in implementations of some array languages such as J, thereby allowing the user to choose the dimension along which a procedure is to be applied. Conceptually, rank is closely related to looping constructs and emulates iteration over a particular dimension in the traditional nested loop program flow typical of ALGOL-inspired languages. J boxing procedures allow a clear illustration of this concept. Starting from a flat array of letters:

```
a. {~ 65 + i. 26
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The rank of this array is 1, as shown by counting its dimensions.

```
# $ a. {~ 65 + i. 26
1
```

Now, we have two available ranks for application: 0 (scalar) and 1. < and " are respectively the boxing and rank operators. Notice the difference in boxing pattern when using different ranks:

```
<"0 a. {~ 65 + i. 26
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
<"1 a. {~ 65 + i. 26
+--------------------------+
|ABCDEFGHIJKLMNOPQRSTUVWXYZ|
+--------------------------+
```

Similarly, this concept extends to any number of dimensions:

```
2 3 $ i. 6
0 1 2
3 4 5
<"1 [ 2 3 $ i. 6
+-----+-----+
|0 1 2|3 4 5|
+-----+-----+
<"2 [ 2 3 $ i. 6
+-----+
|0 1 2|
|3 4 5|
+-----+
```

This feature is extensively used throughout our implementation and forms the basis of a rank-polymorphic unification procedure that will be described below.

Metaprogramming. We use J's first-class quote and eval operators to delay and force evaluation. An internal value can be automatically converted to its string representation by the runtime using the `(5!:5)` function, while its counterpart `".` evaluates J expressions in string form. Unlike Racket which has to interact with the lexical context to recover variable bindings when using `eval`, J metaprogramming is much more primitive and done purely through evaluation of strings without any notion of the surrounding environment.

Fixed Point and Recursion. The `^:_` primitive allows recursion up to a fixed point, which is critical in our unification algorithm implementation heavily inspired by Robinson's classic work [25]. J does not provide tail-call optimization and recursion may result in stack overflow. While transforming the reference implementation to use iteration instead of recursion is an option, the J `F.` operator family provides very flexible folding capabilities used to circumvent restrictions on recursion.

Tree Manipulation. J differs from its more popular sibling APL by providing tree processing primitives. Tree manipulation in array languages is considered a major weakness, but this has recently improved significantly and has become a point of major research interest within the array language community [16]. J's preferred tree representation is nested arrays of boxed elements. The `L:` and `S:` primitives perform left pre-order tree traversal and respectively apply a function in place or apply and collect leaves up to a user-defined tree height. The intrinsic depth-first search capabilities of those primitives are of particular interest in the context of logic programming. The primitive `{::` forms the basis of our unification algorithm proper (see below). `{::` takes a J tree and returns a tree of identical shape where leaves have been replaced by paths to those leaves in left preorder traversal, as exposed below:

```
('a' ;< 'c';'d');<('b' ;< 'e';'f')
+---------+---------+
|+-+-----+|+-+-----+|
||a|+-+-+|||b|+-+-+||
|| ||c|d|||| ||e|f|||
|| |+-+-+||| |+-+-+||
|+-+-----+|+-+-----+|
+---------+---------+
```

```
{:: ('a' ;< 'c';'d');<('b' ;< 'e';'f')
+-------------------------+-------------------------+
|+-----+-----------------+|+-----+-----------------+|
||+-+-+|+-------+-------+|||+-+-+|+-------+-------+||
|||0|0|||+-+-+-+|+-+-+-+|||||1|0|||+-+-+-+|+-+-+-+|||
||+-+-+|||0|1|0||0|1|1|||||+-+-+|||1|1|0||1|1|1||||
||     ||+-+-+-+|+-+-+-+||||     ||+-+-+-+|+-+-+-+|||
||     |+-------+-------+|||     |+-------+-------+||
|+-----+-----------------+|+-----+-----------------+|
+-------------------------+-------------------------+
```

MICROKANREN IMPLEMENTATION

This section describes the J implementation of the core MicroKanren operators with some MiniKanren extensions. We modified these operators' names to accordance with J's restrictions on identifiers (table 1).

Table 1. J procedure names mapping to the reference implementation.

Scheme	J
==	equ
call/fresh	fsh
disj	dis
conj	con
append	app
append-map	apm
var?	var
find	get
occurs?	occ
ext-s	ext
unify	uni
run	run

The reference implementation `define-relation` operator is replaced by quotation at the top of every relation definition as described in Hemann *et al.* [12]. Just like for the Racket version, variables are defined as scalar natural number values. In this section we discuss some of the implementation's more interesting portions. The term "list" in the context of our implementation denotes a flat array. The full code is available in appendix A at the end of this paper as well as online. [1] The J version used was j901/j64avx2/linux release e, with library version 9.01.23.

Variable Scoping and Substitution

MicroKanren uses a substitution table for representing the *most general unifier* from mathematical logic. Ideally, this should be implemented using a hash table or a datastructure with similar insertion and retrieval characteristics although association lists may be used instead. Associative arrays are traditional and often used in APL descendents. However, we will use a flat boxed array as described in Brown & Guerreiro for our substitutions [3]. First, this removes some levels of indirections in our code when walking the substitution and follows the minimalistic spirit of MicroKanren. Second and most importantly, variable scoping is defined by the length of the substitution array as the semi-open interval [0 ; length), as below.

```
fsh =: ,<@#
fsh 3 ; 1 ; 2 ; 'a'
+-+-+-+-+-+
|3|1|2|a|4|
+-+-+-+-+-+
```

The above code block shows the application of `fsh` to an array of length 4. When using this array as a substitution list, scoping rules respectively allow variables 0 through 3 in the original state and 0 through 4 after extension by `fsh`. Position 0 contains the value 3 and points to this location in the list, which itself contains the character *a*. Variable 0 is therfore bound to the value *a*. Variables 1 and 2 are not bound and hence point to themselves. While this system works well in practice, it does not respect lexical scoping since the introduction of a variable causes it to stay in scope wherever its associated substitution table is found. For example, the variables in scope will be passed along when the substitution is returned from a procedure.

Unification

The Racket version of unification relies on tree recursion, which is an obvious choice in this context. Since J does not offer advanced recursion facilities, we instead reuse the tree primitives depth-first search characteristics to yield an iterative algorithm in an attempt to compensate for J's weaknesses.

Walking The Substitution. `get` searches for the value of a variable in a substitution. Similarly to the reference implementation, `get` is the identity function in case the argument is not found in the substitution. The definition uses the `u^:v y` control flow operator where the predicate `v` passes its argument `y` to the function `u` on success. In case of failure, the expression is the identity function and `y` is returned. This gives us "backtracking for free" in a single operator. `{::` retrieves the selected element from the substitution. This pointer-directed walk across the substitution array is repeated until a fixed point is reached using `^:_`, thereby allowing a terse expression of our desired behaviour.

```
get =: {::~^:(var@])"(_ 0)^:_
```

The Occurs Check. The occurs check is non-optional in Kanren-style unification. This is again done using tree recursion in Racket. Here, we rely on J's `S:` operator to avoid writing the tree traversal ourselves. The `occ` predicate is implemented by first checking whether the top-level elements to be compared are identical, in which case we immediately fail since by definition identical terms do not include each other. We then traverse the tree and flatten it to a list of leaves using `< S: 0`. Membership is then tested using `e.`.

```
occ =: (e. < S: 0)`0:@.-:
```

[1] [https://github.com/Rscho314/mk]

Extending The Substitution. The orginal Racket code performs an occurs check and extends the state on failure of the predicate. `ext` is more involved because it operates on the collected set of candidate substitutions it receives from `uni`, using the largest possible fragment of the candidate substitution set on each iteration. Due to our non-lexical scoping system, we have to verify whether the substitution already contains incompatible values and abort if such is the case. This is done by grounding the variables in the tree `y` according to the existing substitution list `x` (`ext`, line 4). `_1` is the error code value (`ext`, line 8). Additionally, duplicate subterms have to be filtered out (`ext`, line 5). Iterations of the while loop insert ground values into the substitution and attempt to ground remaining variables until only free variables (or nothing) remains. This is done by following a chain of pointers from the substitution as mentioned above (`ext`, lines 10-15). Finally, the remaining free variables are inserted into the substitution (`ext`, line 17). The extension of the substitution list involves copying and no sharing of the substitution table occurs, unlike Racket where the runtime can use structure-sharing to preserve space-efficiency. In principle nothing forbids similar sharing in array languages, but most representatives of the paradigm use mutable arrays and rely on interpreters of relatively simple structure to perform optimizations through array-specific instructions such as single instruction multiple data (SIMD) intrinsics.

```
 1  ext =: 4 : 0
 2   while. 1
 3    do.
 4     y =. x&get L:0 y
 5     y =. ~. (#~ (i.@# < i.&1"1@(-:"1/ |."1)~))~ y
 6     if. '' -: y do. x return. end.
 7     isvar =. > var &.> y
 8     if. +./ (occ/"1 y) , (+:/"1 isvar) do. _1 return. end.
 9     candidates =. (*:/"1 isvar) # y {"1~ |."1 isvar
10     if. '' -.@-: , candidates
11      do.
12       x =. ({:"1 candidates) (>@{."1 candidates) } x
13       y =. y -. candidates
14     else. break.
15     end.
16   end.
17   if. '' -.@-: , y do. ({:"1 y) (>@{."1 y) } x  else. x end.
18  )
```

Unification. Rank polymorphism in unification is implemented by boxing arguments `u` and `v` according to the left (`{.x`) and right ranks (`{:x`) (`uni`, lines 4-5). This step is actually independent of the rest of the algorithm which could be replaced by a totally different one without impacting rank polymorphism capabilities. Regarding unification proper, the key to unification using mostly flat array operations is the `{::` (map) J primitive. The explanations and examples regarding `>"` and `{::` found above ("Choosing J" section) are relevant here. `{::` allows us to partition the path tree between variables, corresponding substitutions and residual paths. The path tree is built from `u` (left) and `v` (right) subtrees and flattened to a list of paths using `< S: 1` (`uni`, line 7-8). We also flatten the original tree to a list of leaves with `S: 0` which allows us to create a boolean mask and extract variables and substitutions paths in `varpaths_candidates` and `substpaths_candidates` (`uni`, line 9-10). Substitution paths are obtained by boolean negation of the first path element of `varpaths_candidates`, which points to the corresponding leaf in the opposing subtree (left or right). This avoids traversing the full tree twice (`uni`, line 10) but some of the resulting paths may not actually exist. Those spurious paths stem from variables having no substitutions, *i.e.* free variables. Non-existing paths are identified by checking whether they exist in the set of all path prefixes and free-variable-spurious-substitution pairs are removed if not found (`uni`, lines 11-13). Filtering free variables yields the `varpaths` and `substpaths` lists. As a penultimate step, residual paths not belonging to substitutions are tested for symmetry or emptiness (`uni`, lines 15-19). Finally, we build a table of variables with their corresponding substitutions. This is then passed to `ext` which performs the actual extension of the substitution list (`uni`, line 21). `uni` errors out if out-of-scope variables are present or if rank-defined reboxing results in an unblanced number of cells.

```
 1  uni =: 2 : 0
 2   (_ _) u uni v y
 3  :
 4   u =. , <"({.x) u
 5   v =. , <"({:x) v
 6   if. (#u) ~: #v do. _1 return. end.
 7   tree =. u;<v
 8   paths =. (<S:1)@{:: tree
 9   varpaths_candidates =. paths #~ var S:0 tree
10   substpaths_candidates =. ((-. &.>@{.) 0} ]) &.> varpaths_candidates
11   prefix_paths =. ;@:(,@(<\) &.>)
12   subst_exists =. e.&(prefix_paths paths) substpaths_candidates
13  'varpaths substpaths' =. subst_exists&# &.> varpaths_candidates ;< substpaths_candidates
14   residual_paths =. paths -. varpaths , substpaths
15   if. -. */ ((-.~ prefix_paths)~ residual_paths) e. prefix_paths substpaths
16    do. if. (0 = 2&|) # {::&tree &.> residual_paths
17         do. if. ~:/ ($~ (2,-:@#))~ {::&tree &.> residual_paths do. _1 return. end.
18        else. _1 return.
19        end.
20   end.
21   y ext {::&tree L:1 varpaths ,. substpaths
22  )
```

At this point, we have all the pieces necessary to implement the equality constraint. As in the reference implementation, this can be done in a straightforward manner by simply unifying ground terms. The result is boxed to correspond to the Kanren custom of returning a list of substitutions even if there only one such substitution. The example below shows the main advantage of our implementation. Here, J's ability to operate seamlessly over multidimensional arrays naturally extends to MicroKanren and allows us to unify two three-dimensional arrays. One array contains the alphabet drawn from the ASCII set while the other contains a sequence of the variables 0 through 25.

```
] abc =. (13 2 1 $ ;/ a. {~ 97+i.26) equ (13 2 1 $ ;/ i.26) fsh^:26 ''
+-----------------------------------------------------+
|+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+|
||a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z||
|+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+|
+-----------------------------------------------------+
```

The shape polymorphism inherent to our unification capabilities arises naturally from the use of J's operators, and therefore prevents unification of arrays of incompatible shapes. In this context though, incompatible does not mean that unification is restricted to terms of identical shape. Through the direct manipulation of rank, we can define which term fragments are to be unified. The example below illustrates this process: the left and right ranks are r espectively `0` and `1`. This allows us to unify each of the two fresh variables `26` and `27` of the left array with an entire row of the `2 X 6` letter array on the right. As is expected from MicroKanren relational semantics, we can now use bound variables from the substitution table to produce an array of any desired shape. `bpro` is syntactic sugar allowing simplified answer projection.

```
(26 27) bpro (3 :'(0 1) ((#y),(>:#y)) equ (2 6 $ > 15 4 0 13 20 19 1 20 19 19 4 17) fsh^:2 y') > abc
+------+------+
|peanut|butter|
+------+------+
```

Disjunction, Conjunction and Further Constraints

`app`, `apm`, `dis` and `con` are almost literal transcriptions of their Racket counterparts. The only modification resides in delayed evaluation handling. J quotation implemented as the `(5!:5)` function results in a string that may later be executed using the `".` primitive. Expressions that are already in string form but require additional quotation prior to execution are obtained using the `quote` function. Furthermore, our promises are boxed so that they can be appended to substitutions, thereby constituting what is called an "immature stream" in Kanrens. Promises are built both in `app` and `apm`, and also at the top level of every relation definition as a replacement to Racket's `define-relation` macro [12]. In the code below, `2&=@(3!:0)@> x` is the equivalent of Racket's `promise?` predicate. line 5 in `app` builds the boxed string promise.

```
1  app =: 4 : 0
2   if. x -: ''
3    do. y
4   elseif. 2&=@(3!:0)@> x
5    do. < '(' , ((5!:5)<'y') , ') app ((3 :' , (quote 0 {:: x) , ')'''')'
6   elseif. do. ({. , (y app~ }.)) x
7   end.
8  )
```

In our J implementation we bypassed the `call/initial-state - take - pull` call chain from the reference implementation and directly implemented `run`. J's `F:` primitive allows us to iterate over the immature stream and collect results as a mature stream all at once. On each iteration, the promise resulting from the previous execution is forced using `".` and the result is collected by taking the head (`{.`) of the new value. The definition of `run` can be found in appendix A.

EXAMPLE GOALS AND BENCHMARKING

Here, we illustrate the execution steps of our implementation on a simple goal. The `fives_and_sixes` goal yields a stream of alternating values up to a user-defined length and uses `run` facilities identical to the reference implementation. `fives_and_sixes` is implemented as the disjunction of subgoals `fives` and `sixes` and shows that goal combinator functionalities similar to the reference implementation are available. Our ability to generate a stream of alternating values illustrates the interleaving search typical of Kanrens and the fact that our program correctly handles infinte recursive goals. As shown below, the execution of a recursive goal yields a pair of a value and a newly built promise. On further execution, the value is added to the result while the promise is forced to yield the following value-promise pair in the sequence.

```
fives_and_sixes fsh ''
10 run fives_and_sixes fsh ''
```

```
+---+-----------------------------------------------------+
|+-+|((<,<6.),<'sixes (,<00)') app ((3 :'fives (,<00)')'')|
||5||                                                     |
|+-+|                                                     |
+---+-----------------------------------------------------+
```

```
+---+---+---+---+---+---+---+---+---+---+
|+-+|+-+|+-+|+-+|+-+|+-+|+-+|+-+|+-+|+-+|
||5|||6|||5|||6|||5|||6|||5|||6|||5|||6||
|+-+|+-+|+-+|+-+|+-+|+-+|+-+|+-+|+-+|+-+|
+---+---+---+---+---+---+---+---+---+---+
```

We will now explore some goals through microbenchmarking which will reveal some of the stronger and weaker points of the J implementation. All benchmarking programs can be consulted in appendix B. Performance was not the main focus in this work. First, let us test the unification of two flat arrays of size *N*, one of which comprises only variables and the other only random float values (goal `floats`, appendix B).

Table 2. flat array of random floats unification benchmark (mean of 10 runs). Racket CS 7.7 on an Intel Core i7-6800K. Time columns are in seconds. Other columns are cumulative (cum.), step-by-step (step) and raw time ratios (Racket *vs.* J).

	J			Racket			Racket vs. J
N	time	cum.	step	time	cum.	step	
4	0.000165	_	_	0	_	_	_
16	0.000242	1.5	1.5	0	_	_	_
64	0.001764	10.7	7.3	0	_	_	_
256	0.022942	139	13	0.0001	_	_	230
1024	0.327932	1987.5	14.3	0.003	_	30	109
4096	4.84869	29386	14.8	0.0521	_	17.4	93
16384	77.4688	469507.9	16	1.1027	_	21.2	70

The reference implementation is much faster but raw running time comparison is not the most interesting element of our benchmarks given the completely different runtime architectures. Results from table 2 suggest that our trivial unification algorithm scales at least as well and perhaps better than the original in this case although J's interpreter is rather simplistic compared to Racket's compiler. This benchmark represents a particular situation that is most favourable to the J program because all variables are immediately unified with a value. In such a situation, the `ext` while loop immediately breaks without iterating as there are no candidate terms to filter, no remaining free variables and the whole array is therefore processed all at once without the need to walk the substitution list. Let us now illustrate the opposite situation. The following benchmark again tests the unification of two flat arrays but this time with a single starting value that propagates to the whole result array through a chain of variables (goal `as`, appendix B).

From the results in table 3, we see that Racket now has a dramatic advantage in speed. The same advantage would likely be seen in step ratios but J is slow enough on this benchmark that further testing of algorithmic scaling goes beyond our available time resources. Those results were expected since the test causes *N* iterations of the interpreted while loop in `ext`, and walking the substitution list is required on each iteration. As an intermediate

Table 3. unification benchmark of two flat arrays comprising only variables at the exception of a single value that propagates to yield a list of size *N* including only repeated instances of the single starting value (mean of 10 runs). Racket CS 7.7 on an Intel Core i7-6800K. Time columns are in seconds. Other columns are cumulative (cum.), step-by-step (step) and raw time ratios (Racket *vs.* J).

	J			Racket			Racket vs. J
N	time	cum.	step	time	cum.	step	
4	0.000823	_	_	0	_	_	_
16	0.003725	4.5	4.5	0	_	_	_
64	0.014833	18	4	0	_	_	_
256	0.22464	273	15.1	0	_	_	_
1024	7.53526	9155.9	33.5	0.0009	_	_	8373
4096	411.842	500415.6	54.7	0.0185	_	20.6	22262

between the programs from tables 2 and 3 additionally mobilizing the `run` facilities in both implementations, let us now benchmark peano numbers generation.

Table 4. peano 0 to *N* generation benchmark (mean of 10 runs). Racket CS 7.7 on an Intel Core i7-6800K. Time columns are in seconds. Other columns are cumulative (cum.), step-by-step (step) and Racket *vs.* J running time ratios.

	J			Racket			Racket vs. J
N	time	cum.	step	time	cum.	step	
4	0.001625	_	_	0	_	_	_
16	0.005917	3.6	3.6	0	_	_	_
64	0.018768	11.5	3.2	0.0004	_	_	46.9
256	0.0874992	53.8	4.7	0.0097	_	24.25	9
1024	0.960714	591.2	11	0.4949	_	51	1.9
4096	26.6484	16399	27.7	58.1528	_	117.5	0.5

The benchmarking results from the `peano` goal in table 4 are more surprising. Although Racket is still much faster on smaller data, J quickly catches up and even surpasses Racket on larger data. Unfortunately, the origin of this advantage remains unclear despite testing (not shown) but reveals that J can be more efficient on some workloads.

RELATED WORK

Logic programming over arrays is not a new idea but has historically enjoyed relatively little interest compared to LISP or Prolog solutions both prioritzing homoiconicity and lists as fundamental datastructure, making them ideal candidates for symbolic applications. In the array language domain some commercial systems include a limited unification system in their library, although array languages have traditionally focused on statistical and numerical computing [1]. Work focused on unification algorithms in the array context was explored by Brown *et al.* in 1987 [3]. By then, a large *corpus* of literature regarding logic capabilities of array languages was already available and those languages were already viewed by some as candidates for the "5th generation" computing project [22]. Following the work of Brown *et al.*, predicate calculus was implemented in APL2 by Engelmann *et al.*, bringing array languages closer to the Prolog world [8]. The newer trend of artificial intelligence based upon statistical rather than symbolic computing was then followed by APL-family representatives, unsurprisingly revealing themselves very capable environments in those array-heavy computation models up to this day [28]. The mixing of array- and logic-based solutions remains a research topic of substantial interest in a variety of fields, from end-user ergonomics to compilation techniques [7, 29]. Conceptually, hybrid reasoning may result in broader capabilities for artificial intelligence systems. In hierarchical control systems such as those found in behaviour-based robotics, the high-level symbolic component can rely on a reactive lower-level statistical (also termed *subsymbolic*) layer. The higher-levels benefit from relaxed time constraints allowing them to perform planning. Hybrid reasoning can result in a model of artificial intelligence capable of reasoning through induction, decuction, abduction and analogy and emulates natural systems [2]. Some recent works suggest that the unification of symbolic and statistical artificial intelligence may perhaps find potential for progress in the array-language world [19]. Indeed, arrays are a *must-have* for statistical computing and easy manipulation of arrays therefore must be available to the user wanting to combine statistical and symbolic reasoning. There are currently a variety of possibilities for programmers wanting to combine logic and array programming ranging from impure (`setarg/3`) or unwieldy (`arg/3`) Prolog idioms to non Turing-completeness *a la* Datalog. Those solutions represent two faces of the same Prolog coin, the former suffering from the usual pain points regarding logical purity and termination guarantees, while the latter imposes important semantic restrictions on the user. Furthermore, the implementation of performant Prolog or similar systems is far from trivial.

LIMITATIONS AND FUTURE PROSPECTS

The limitations of our system are in part those of MicroKanren in that only the equality constraint is implemented and most interesting logic problems require other more expressive constraints. Unfortunately, syntactic constraints make the elaboration of programs in our system rather fastidious and will require the addition of syntactic sugar to ease the writing of additional constraints. A second limitation is that as in the original MicroKanren we abandon most of the arithmetic capabilities of the host language by using natural numbers as variables. This is an opportunity to implement a relational arithmetic suite in the spirit of Kiselyov *et al.* [18]. Finally, the rank-polymorphism of our unification process is a natural extension benefiting from the host language's capabilities and is actually completely distinct from the unification algorithm itself. In its current form, our unification procedure is expected to be slow with regard to the state of the art and should perhaps be replaced by a program with better time and space complexity as described for example in Paterson & Wegman [24]. Interestingly, performant unification algorithms aim at the constitution of equivalence classes and do not use a "find" procedure. Such limitation of indirections in our program would perhaps allow a more vector-friendly implementation. However, some research suggests that the theoretical advantages of alternatives to Robinson's algorithm are not always realized in practice [14]. In this context, a first step will be the developement of an appropriate benchmarking framework geared towards vectorization opportunities. While our current implementation does not offer the features of mature logic environments, it delivers on the promise of a lightweight and logically pure basis for further exploration. Indeed, porting MicroKanren to an alternative datastructure basis might result in a new program writing style deriving from different tradeoffs imposed from the underlying arrays. This has not been explored in this paper, as the focus was on showing that the core language could be adapted. By implementing Kanren on top of an array language, we get close to a relational database management system and its traditional command language: SQL. MiniKanren-inspired systems for database querying have been deployed in production environments with success [4, 27]. Although extensions have recently made SQL Turing-complete, its impure semantics do not allow it to be considered a strict implementation of either predicate calculus or relational algebra. In contrast, MicroKanren has very clean semantics and holds potential for user-defined extensions, parallelism and constraint logic programming. The parallel drawn to database systems does not stop here: J includes Jd, an integrated column-store database. This is an opportunity for the implementation of a tabled version of MicroKanren based on Byrd's thesis that hopefully would improve both efficiency and termination guarantees of the system [6]. A second opportunity offered by Jd is to use it as a constraint store to implement constraint MicroKanren as described by Hemann *et al.* [11]. This would bring us closer to systems such as XSB Prolog while remaining in the purely relational context [26]. Another potential improvement would be constructive negation, which was recently described for MiniKanren and especially in combination with tabling would take us closer to supporting stable model semantics in Kanrens [23]. Finally, running a MicroKanren-inspired system on the GPU for massive parallelism is an encouraging research perspective that was already considered in Eric Holk's Harlan project which initially implemented a MiniKanren type checker and region inferencer [15]. In the array language world, the APL Co-dfns compiler recently brought us closer to running general programs on the GPU. However, Co-dfns does not currently provide delayed evaluation capabilities nor does it support nested arrays and alternatives will be required if the microKanren path is to be pursued on this platform [17]. Running on top of high-level GPU languages such as Futhark might also be considered, especially since prior art exists for this use case in the TAIL APL compiler [13].

CONCLUSION

This work led us to the fact that J has many features making it ideal for the developement of a logic library, and that unification can be made rank-polymorphic by using characteristics of the array language paradigm as a stepping stone. While there are many enticing future perspectives in the line of research combining array and logic programming, many of those are open research problems. Despite that, this work provides a starting point for more involved forays into array-based pure relational programming.

ACKNOWLEDGMENTS

The author would like to express his gratitude to Dr. J. Hemann for his kindness and proofreading of the manuscript. Thanks also go to Dr. A. Hsu for his encouragments and to all members of the programming@jsoftware.com mailing list and r/apljk subreddit and especially Henry Rich, Julien Fondren and Raul Miller for their help with J language concepts. Finally, the author thanks the reviewers of the ICFP 2020 MiniKanren workshop for their useful advice.

REFERENCES

[1] [n.d.]. Dyalog APL - Unification of Expressions (Visited May 11, 2020). https://dfns.dyalog.com/n_unify.htm.
[2] J.S. Albus. 1993. A Reference Model Architecture for Intelligent Systems Design. In *An Introduction to Intelligent and Autonomous Control*. Kluwer Academic Publishers, 27–56.
[3] James A. Brown and Ramiro Guerreiro. 1987. APL2 Implementations of Unification. In *Proceedings of the International Conference on APL: APL in Transition (APL '87)*. Association for Computing Machinery, Dallas, Texas, USA, 216–225. https://doi.org/10.1145/28315.28342
[4] Craig Brozefsky. 2013. SQL and Core.Logic Killed My ORM ; ClojureWest 2013 ; (https://www.infoq.com/presentations/Core-logic-SQL-ORM/).
[5] William Byrd. 2017. Relational Interpreters, Program Synthesis, and Barliman: Strange, Beautiful, and Possibly Useful ; CodeMesh 2017 London ; (https://codemesh.io/codemesh2017/william-e-byrd).
[6] William E. Byrd. 2009. *Relational Programming in Minikanren: Techniques, Applications, and Implementations.* Ph.D. Dissertation. Indiana University, USA. Advisor(s) Friedman, Daniel P.
[7] Philip T. Cox and Patrick Nicholson. 2008. Unification of Arrays in Spreadsheets with Logic Programming. In *Practical Aspects of Declarative Languages (Lecture Notes in Computer Science)*, Paul Hudak and David S. Warren (Eds.). Springer, Berlin, Heidelberg, 100–115. https://doi.org/10.1007/978-3-540-77442-6_8
[8] U. Engelmann, Th. Gerneth, and H. P. Meinzer. 1989. Implementation of Predicate Logic in APL2. *ACM SIGAPL APL Quote Quad* 19, 4 (July 1989), 124–128. https://doi.org/10.1145/75145.75162
[9] Jason Hemann and Daniel P Friedman. 2013. μKanren: A Minimal Functional Core for Relational Programming. In *Scheme and Functional Programming Workshop*, Vol. 2013.
[10] Jason Hemann and Daniel P Friedman. 2015. How to Be a Good Host: miniKanren as a Case Study ; Curry On 2015 Prague ; (https://www.curry-on.org/2015/sessions/how-to-be-a-good-host-minikanren-as-a-case-study.html).

[11] Jason Hemann and Daniel P. Friedman. 2017. A Framework for Extending microKanren with Constraints. *Electronic Proceedings in Theoretical Computer Science* 234 (Jan. 2017), 135–149. https://doi.org/10.4204/EPTCS.234.10 arXiv:1701.00633
[12] Jason Hemann, Daniel P. Friedman, William E. Byrd, and Matthew Might. 2016. A Small Embedding of Logic Programming with a Simple Complete Search. In *Proceedings of the 12th Symposium on Dynamic Languages - DLS 2016*. ACM Press, Amsterdam, Netherlands, 96–107. https://doi.org/10.1145/2989225.2989230
[13] Troels Henriksen, Martin Dybdal, Henrik Urms, Anna Sofie Kiehn, Daniel Gavin, Hjalte Abelskov, Martin Elsman, and Cosmin Oancea. 2016. APL on GPUs: A TAIL from the Past, Scribbled in Futhark. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing (FHPC 2016)*. Association for Computing Machinery, Nara, Japan, 38–43. https://doi.org/10.1145/2975991.2975997
[14] Kryštof Hoder and Andrei Voronkov. 2009. Comparing Unification Algorithms in First-Order Theorem Proving. In *KI 2009: Advances in Artificial Intelligence (Lecture Notes in Computer Science)*, Bärbel Mertsching, Marcus Hund, and Zaheer Aziz (Eds.). Springer, Berlin, Heidelberg, 435–443. https://doi.org/10.1007/978-3-642-04617-9_55
[15] Eric Holk, William E Byrd, Nilesh Mahajan, Jeremiah Willcock, Arun Chauhan, and Andrew Lumsdaine. 2011. Declarative Parallel Programming for GPUs. 297–304.
[16] Aaron W. Hsu. 2016. The Key to a Data Parallel Compiler. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2016)*. Association for Computing Machinery, Santa Barbara, CA, USA, 32–40. https://doi.org/10.1145/2935323.2935331
[17] Aaron W. Hsu. 2019. A Data Parallel Compiler Hosted on the GPU. (Nov. 2019).
[18] Oleg Kiselyov, William E Byrd, Daniel P Friedman, and Chung-chieh Shan. 2008. Pure, Declarative, and Constructive Arithmetic Relations (Declarative Pearl). In *International Symposium on Functional and Logic Programming*. Springer, 64–80.
[19] Ryosuke Kojima and Taisuke Sato. 2019. A Tensorized Logic Programming Language for Large-Scale Data. *arXiv:1901.08548 [cs]* (Jan. 2019). arXiv:1901.08548 [cs]
[20] Dmitrii Kosarev and Dmitry Boulytchev. 2018. Typed Embedding of a Relational Language in OCaml. *Electronic Proceedings in Theoretical Computer Science* 285 (Dec. 2018), 1–22. https://doi.org/10.4204/EPTCS.285.1 arXiv:1805.11006
[21] Robert Kowalski and Steve Smoliar. 1982. Logic for Problem Solving. *ACM SIGSOFT Software Engineering Notes* 7, 2 (April 1982), 61–62. https://doi.org/10.1145/1005937.1005947
[22] C. D. McCrosky, J. J. Glasgow, and M. A. Jenkins. 1984. Nial: A Candidate Language for Fifth Generation Computer Systems. In *Proceedings of the 1984 Annual Conference of the ACM on The Fifth Generation Challenge (ACM '84)*. Association for Computing Machinery, New York, NY, USA, 157–166. https://doi.org/10.1145/800171.809618
[23] Evgenii Moiseenko. 2019. Constructive Negation for MiniKanren. https://www.semanticscholar.org/paper/Constructive-Negation-for-MiniKanren-Moiseenko/1b7fd5635770759232f3d88682c4d5bb7b5a7b54.
[24] M. S. Paterson and M. N. Wegman. 1978. Linear Unification. *J. Comput. System Sci.* 16, 2 (April 1978), 158–167. https://doi.org/10.1016/0022-0000(78)90043-0
[25] J. A. Robinson. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 1 (Jan. 1965), 23–41. https://doi.org/10.1145/321250.321253
[26] Konstantinos Sagonas, Terrance Swift, and David S. Warren. 1994. XSB as an Efficient Deductive Database Engine. In *In Proceedings of the ACM SIGMOD International Conference on the Management of Data*. ACM Press, 442–453.
[27] Ryan Senior. 2012. Practical Core.Logic ; ClojureWest 2012 ; (https://www.infoq.com/presentations/core-logic/).
[28] Artjoms Šinkarovs, Robert Bernecky, and Sven-Bodo Scholz. 2019. Convolutional Neural Networks in APL. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY 2019)*. Association for Computing Machinery, Phoenix, AZ, USA, 69–79. https://doi.org/10.1145/3315454.3329960
[29] Justin Slepak, Panagiotis Manolios, and Olin Shivers. 2018. Rank Polymorphism Viewed as a Constraint Problem. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2018)*. Association for Computing Machinery, Philadelphia, PA, USA, 34–41. https://doi.org/10.1145/3219753.3219758

APPENDIX A: MICROKANREN SOURCE

```
var =: -.@#@$*.e.&4 1@(3!:0)
get =: {::~^:(var@])"(_ 0)^:_
occ =: (e. < S: 0)`0:@.-:
ext =: 4 : 0
while. 1
 do.
```

```
  y =. x&get L:0 y
  y =. ~. (#~ (i.@# < i.&1"1@(-:"1/ |."1)~))~ y
  if. '' -: y do. x return. end.
  isvar =. > var &.> y
  if. +./ (occ/"1 y) , (+:/"1 isvar) do. _1 return. end.
  candidates =. (*:/"1 isvar) # y {"1~ |."1 isvar
  if. '' -.@-: , candidates
   do.
    x =. ({:"1 candidates) (>@{."1 candidates) } x
    y =. y -. candidates
  else. break.
  end.
end.
if. '' -.@-: , y do. ({:"1 y) (>@{."1 y) } x  else. x end.
)
uni =: 2 : 0
 (_ _) u uni v y
:
 u =. , <"({.x) u
 v =. , <"({:x) v
 if. (#u) ~: #v do. _1 return. end.
 tree =. u;<v
 paths =. (<S:1)@{:: tree
 varpaths_candidates =. paths #~ var S:0 tree
 substpaths_candidates =. ((-. &.>@{.) 0} ]) &.> varpaths_candidates
 prefix_paths =. ;@:(,@(<\) &.>)
 subst_exists =. e.&(prefix_paths paths) substpaths_candidates
 'varpaths substpaths'=.subst_exists&#&.>varpaths_candidates;<substpaths_candidates
 residual_paths =. paths -. varpaths , substpaths
 NB.check if residual values are inside a substitution
 if. -. *./ ((-.~ prefix_paths)~ residual_paths) e. prefix_paths substpaths
 NB.if not inside a substitution, check if non-identical and if so, fail.
  do. if. (0 = 2&|) # {::&tree &.> residual_paths
       do. if. ~:/ ($~ (2,-:@#))~ {::&tree &.> residual_paths do. _1 return. end.
      else. _1 return.
      end.
 end.
 y ext {::&tree L:1 varpaths ,. substpaths
)
equ =: 2 : 0
 (_ _) u equ v y
:
 < x (y get u) uni (y get v) y
)
fsh =: ,<@#
```

```
app =: 4 : 0
 if. x -: ''
  do. y
 elseif. 2&=@(3!:0)@> x
  do. < '(' , ((5!:5)<'y') , ') app ((3 :' , (quote 0 {:: x) , ')'''')'
 elseif. do. ({. , (y app~ }.)) x
 end.
)
apm =: 2 : 0
 if. u -: ''
  do. ''
 elseif. 2&=@(3!:0)@> u
  do. < '((3 :' , (quote@> u) , ')'''') apm (' , ((5!:5)<'v') , ')'
 elseif. do. (v (>u)) app (}.u) apm v
 end.
)
dis =: 2 : '(u y) app (v y)'
con =: 2 : '(u y) apm v'
runhelper =: (".@(1&{::) , (<: &.>@{:)) [ _2&Z:@-.@*@(2&{::)
run =: 2 : 0
 if. u > 1
  do. 'initval initprom' =. v y
      initval ; runhelper F: {. _ ; initprom ; < <: u
 elseif. u = 1 do. {. v y
 elseif. u = 0 do. ''
 elseif. do. _1
 end.
:
 if. u > 1
  do. 'initval initprom' =. v y
      (x&{ &.> initval) ; (x&{ &.>) runhelper F: {. _ ; initprom ; < <: u
 elseif. u = 1 do. {. v y
 elseif. u = 0 do. ''
 elseif. do. _1
 end.
)
bpro =: 4 : '(>y)&get L:0 x&{ > y'
upro =: 4 : '((>y)&get L:0)@> x&{ > y'
```

APPENDIX B: BENCHMARKING CODE

J

```
floats =: (3 : '(0 0) y uni (i. # y) fsh^:(#y) '''' ')
floats_perf =: 3 : 0
 n =. */\ 4 #~ y
 data =. (?@#&0) &.> ;/ n
 t =. (x:!.0)(6!:2)@> 'floats '&,@":@(?@#&0) &.> ;/ n
 ct =. _ , */\ (x:2) %~/\ ; t
 st =. _ , (x:2) %~/\ ; t
 legend =. 'N';'time [s]';'cumulative time ratio';'step time ratio'
 legend , <"0 n ,. t ,. ct ,. st
)
floats_perf 7
as =: 3 : '(0 0) ((<''a'') , ;/ i. <: y) uni (;/ i. y) (;/ i. y)'
as_perf =: 3 : 0
 n =. */\ 4 #~ y
 t =. (x:!.0) (6!:2)@> ('as ' , ":) &.> ;/ n
 ct =. _ , */\ (x:2) %~/\ ; t
 st =. _ , (x:2) %~/\ ; t
 legend =. 'N';'time [s]';'cumulative time ratio';'step time ratio'
 legend , <"0 n ,. t ,. ct ,. st
)
as_perf 6
peano =: 3 : 0
(3 :'''z'' equ (<:#y) y') dis ((3 :'(2-~#y) equ (''s'';<:@#y) y') con
(3 : '<''peano ('',((5!:5)<''y''),'')''')@fsh) y
)
peano_perf =: 3 : 0
 n =. */\ 4 #~ y
 t =. (x:!.0) (6!:2)@> (' run peano fsh ''''',~":) &.> ;/ n
 ct =. _ , */\ (x:2) %~/\ ; t
 st =. _ , (x:2) %~/\ ; t
 legend =. 'N';'time [s]';'cumulative time ratio';'step time ratio'
 legend , <"0 n ,. t ,. ct ,. st
)
peano_perf 6
```

Racket

```
(require microkanren
         atomichron
         racket/flonum)
(define (powers-of-n n iterations)
  (map (lambda (exponent) (expt n (add1 exponent))) (build-list iterations values)))
(define (gensym-list n)
  (build-list n (lambda (x) (gensym))))
(define (random-float-list n)
  (define rg (make-pseudo-random-generator))
  (for/list ([i (build-list n values)]) (flrandom rg)))
(define (time-in-seconds benchmark-times)
  (map (lambda (n) (exact->inexact (/ n (expt 10 9)))) benchmark-times))
(define (random-floats-benchmark l r)
  (make-microbenchmark
   #:name 'random-floats-benchmark
   #:iterations 1
   #:microexpression-iterations 10
   #:microexpression-builder
   (lambda (iteration)
     (make-microexpression #:thunk (lambda () (unify l r '()))))))
(define float-times
    (map
     (lambda (x)
       (let ([left (random-float-list x)]
             [right (build-list x values)])
         (microbenchmark-result-average-real-nanoseconds
          (microbenchmark-run! (random-floats-benchmark left right)))))
     (powers-of-n 4 7)))
(begin (display "time in seconds (random floats): ") (time-in-seconds float-times))
(define (as-benchmark l r)
  (make-microbenchmark
   #:name 'as-benchmark
   #:iterations 1
   #:microexpression-iterations 10
   #:microexpression-builder
   (lambda (iteration)
     (make-microexpression #:thunk (lambda () (unify l r '()))))))
(define as-times
  (map
   (lambda (x)
     (let ([left (cons #\a (build-list (sub1 x) values))]
           [right (build-list x values)])
       (microbenchmark-result-average-real-nanoseconds
 (microbenchmark-run! (as-benchmark left right)))))
   (powers-of-n 4 7)))
(begin (display "time in seconds (list of as): ") (time-in-seconds as-times))
(define (peano-benchmark n)
  (make-microbenchmark
   #:name 'peano-benchmark
   #:iterations 1
   #:microexpression-iterations 10
   #:microexpression-builder
   (lambda (iteration)
     (make-microexpression #:thunk (lambda () (run n (q) (peano q)))))))

(define peano-times (map (lambda (x) (microbenchmark-result-average-real-nanoseconds
  (microbenchmark-run!(peano-benchmark x)))) (powers-of-n 2 3)))
 (begin (display "time in seconds (peano): ") (time-in-seconds peano-times))
```