


# Bi-Directional DSL Transformation Using miniKanren

Philipp Körner 

Heinrich Heine University Düsseldorf — Faculty of Mathematics and Natural Science  
— Department of Computer Science — 40225 Düsseldorf, Germany  
`{p.koerner}@hhu.de`

**Abstract.** The *lisb* library embeds the B specification language in Clojure and uses three representations: an internal DSL, an intermediate representation (IR) and a Java AST. To change between representations, three uni-directional translations are implemented. In this paper, we use a miniKanren implementation to obtain a new bi-directional translation between the internal DSL and the IR.

The resulting code is almost 90 % shorter than the reference implementation. In particular, a data base of all syntactical elements was created that can be re-used, e.g., to generate documentation. However, the code is less readable for Clojure developers. Our experience demonstrates that declarative techniques can indeed be applied in real-world projects, especially in tools supporting DSLs. At the same time, entry barriers might be too high for the general audience. Especially tool support, tighter integration with general-purpose programming languages and documentation should be improved to be attractive for non-experts.

**Keywords:** miniKanren · DSLs · Translation · Clojure

## 1 Introduction

The *lisb* library [11] provides an embedding of B [1], a formal language for state-based specifications, in Clojure [9]. The goal of *lisb* is to express or generate mathematical constraints and formal state-machines in Clojure. *lisb* offers different representations of constraints, e.g.,  $x = 1 + 2$  can be represented in:

- an *internal DSL*, which is more suitable for humans to read and write, as `(b (= :x (+ 1 2)))`;
- an *intermediate representation (IR)*, which is less readable, but more suitable for programmatic transformation, as the nested map literal `{:tag :equals, :left :x, :right {:tag :add, :nums (1 2)}}`;
- a collection of automatically generated AST nodes of PROB's [14] parser suite, used to interact with the Java API of PROB [12], a constraint solver and model checker for B.

A macro can re-write the internal DSL to expressions which evaluate to the IR. The IR can be translated into the Java AST, which can be transformed back into the internal DSL.

In this paper, we report on our experience using Clojure’s contrib library *core.logic*, an implementation of miniKanren [6], while re-implementing the transformation from the internal DSL into the IR. This translation is extremely simple, especially if done in a full Prolog-style. At the beginning, we only had limited knowledge of *core.logic*. We chose it because of the promise of Prolog-like programming as a Clojure library and its seemingly nice integration with Clojure and its data structures (such as maps, sets and metadata). Nonetheless, we hoped to reap benefits similar to the strengths of Prolog [13], such as:

- obtaining a logical relation between the DSL and the IR that can be executed in both directions (transforming the DSL into the IR and vice versa);
- organising the translation rules in form of a data base that can be used for manual and automatic verification as well as for documenting the DSL;
- maintaining only *one* relation instead of more than 150 individual functions, especially for enhancements such as linking DSL code with the final output;
- overcoming limitations of the macro-based approach where the underlying JVM method bytecode limit restricts the size of the DSL input;
- gaining insights for possible future transformations (between IR and the Java AST, user DSLs and the internal DSL and IR-to-IR-transformations).

This paper is structured as follows: we give a brief overview over miniKanren and the *core.logic* library as well as the *lisb* library and the two representations we want to transform between in Section 2. The new implementation and our solutions to issues that came up during development are outlined in Section 3. We discuss the resulting code and our experience during development in Section 4 and draw our conclusion in Section 5.

## 2 Background

Below, we briefly highlight relevant points regarding the used libraries.

### 2.1 miniKanren and the *core.logic* Library

miniKanren [6] is a domain-specific language for logic programming, initially developed in Scheme. The idea is to extend *functional* programming, where each function has exactly one return value, to *relational* programming, where relations return true iff the input values are contained in that relation (like Prolog predicates). For this, miniKanren allows introducing logical variables and implements a backtracking search<sup>1</sup>. The miniKanren core is very compact consisting only of four operators (`run` — the main entry point to change to miniKanren’s semantics —, as well as the equivalents of `==`, `fresh` and `conde` described below). However, it is intended to be extensible (e.g., by tabling or nominal logic).

<sup>1</sup>“The core miniKanren implementation [...] uses a stream-based interleaving search strategy [which] makes it difficult to exactly characterize the search behavior, and therefore the order in which miniKanren produces answers” [6, Ch. 17.1]

The *core.logic* library<sup>2</sup> is an implementation of miniKanren<sup>3</sup> in Clojure [9], a functional Lisp that runs on the JVM. With some extensions, the project describes itself as offering “Prolog-like relational programming, constraint logic programming, and nominal logic programming for Clojure”. For this paper, we make use of:

- `==`, which tries to unify two logical terms (like `=` in Prolog),
- `fresh`, which introduces new, unbound logical variables in a scope (which automatically happens in Prolog by introducing a variable),
- `conde`, which creates a logical disjunction of clauses (like Prolog’s “;”),
- `conso`, which can be used to construct and destruct lists (`L = [H|T]` in Prolog),
- `matche`, a pattern matching macro (included in Prolog’s unification),
- the non-relational function `project` that allows extracting the value from a logical variable in order to interact with it via arbitrary Clojure functions.

## 2.2 *lisp* Library

The *lisp* library embeds the B specification language<sup>4</sup> [1] in Clojure. The name “B” is used both to refer to the language and a formal methodology to obtain correct-by-construction software: Typically, state machines are written and proven correctly. Details of *how* the computation shall work are added in refinement steps, which are linked to the more abstract version via proof. B has also been used for data validation projects [5], where it is checked that data satisfies complex properties described in first-order logic and set-theoretical constructs. One tool supporting the B method is PROB, a model checker, constraint solver and animator for B.

The goal of the *lisp* library is to provide a tool to easily write constraints or state machines in Clojure or programmatically generate them from external data sources. Below, we informally outline its DSL for B, the underlying IR, and the existing, functional translation between those two.

**B DSL** The internal B DSL is designed to feel natural to Clojure programmers, yet stay close to the original B constructs. This results in some trade-offs where the DSL is not uniform:

- Literals, i.e., keywords (to represent logical variables), strings and numbers can be directly written.
- Constants, such as the set of natural numbers, are symbols.
- Most operators feel like regular function calls, e.g., `(+ 1 2)`.
- Operators may have optional arguments, e.g., in `(if cond then else)`, the `else` branch is optional, whereas others have a variadic number of parameters, e.g., `(+ 1 2 3 4 ...)`.

<sup>2</sup> <https://github.com/clojure/core.logic>

<sup>3</sup> It even was described as the *most popular* implementation by Byrd.

<sup>4</sup> For the understanding of this article, no knowledge of B is required.

- Sequence data types (vectors and lists) are not interchangeable: e.g., the expression `(for-all [:x :y] ...)` introduces the new variables `:x` and `:y` in a vector (and explicitly not a list).
- There is some syntactic sugar, e.g., we expect that the `for-all` operator (`(for-all [:x] (=> premise conclusion))`) has an implication as its body, but also allow splicing the premise and conclusion into two arguments (`(for-all [:x] premise conclusion)`). Further, there are aliases for operators which align with Clojure function names (e.g., a `contains?` variant of the `member?` predicate that aligns with Clojure naming and argument order);
- There are some further special cases in the DSL which we will not cover for the sake of brevity.

**Intermediate Representation** The IR in *lisp* aligns (mostly) with PROB’s Java AST. In contrast to the AST objects, however, the data representation allows duplicating and moving sub-trees arbitrarily. A few re-writes are performed before transforming it into the Java AST, e.g., enforcing that arithmetic operators are binary. As an example, the IR data for the B DSL snippet `(for-all [:x] (member? :x nat-set) (<= :x 0))` is as follows:

```
{:tag :for-all, :ids [:x],
 :implication {:tag :implication,
               :preds ({:tag :member, :elem :x, :set {:tag :nat-set}}
                       {:tag :less-equals, :nums (:x 0)})}}
```

Each map has a `:tag` that defines the operator. The arguments are stored under keys which depend on the operator (e.g., `:implication`, `:elem` and `:set`, or `:nums` in the example above). The differing keys serve two functionalities: aside from storing arguments, they state a type that can be verified by Clojure’s Spec [15, Ch. 5] library (which provides an optional, ad-hoc type system).

**Reference Implementation** In order to transform the B DSL into the intermediate representation, we initially implemented a functional version (referred to as *reference implementation* below). The reference implementation makes use of a macro, a programmatic source-to-source translation. The macro inserts the DSL code as a body of a `let`-expression: symbols for operators (such as `=` or `for-all`) are (re-)bound to *lisp* library functions that generate an IR snippet. This macro has the following form:

```
(defmacro b [body] ... some pre-processing ...
  `(let [~'+ b+, ~'- b-, ~'for-all bfor-all, ...]
       ~pre-processed-body))
```

The functions that generate the corresponding IR snippets all are similar and small. They simply take all arguments and store them at the correct position in a map. Due to the evaluation mechanism of Clojure, arguments are already evaluated to their corresponding IR. For example, the function `b+` that returns the IR for an addition is defined as follows:

```
(defn b+ [& nums] {:tag :add, :nums nums})
```

The reference implementation consists of about 170 functions for the operators and a few functions for pre-processing special cases in the DSL.

### 3 Implementation

In this section, we outline the important parts of our new translation. The goal is to transform the B DSL, with expressions such as `(b (= :x (+ 1 2)))`, into an intermediate representation of constraints or state machines, such as `{:tag :equals, :left :x, :right {:tag :add, :nums (1 2)}}`.

We will focus on the general case<sup>5</sup> of the translating relation (Fig. 1) and — for the sake of brevity — will ignore the handling of special cases. Below, we will directly refer to this listing by citing the relevant line numbers as we discuss the covered aspects.

Figure 2 tries to approximate this translation relation in Prolog. This version differs in some aspects: Most Prolog systems do not support maps<sup>6</sup>, so the IR argument is a list of tuples representing key-value pairs instead. We assume that this list of pairs is ordered correctly — a map would not have an order one could rely on (see Section 3.3 for details). Finally, the representation of the data base is not idiomatic in Prolog. Instead, one could specify the same variable twice (like `rule(["for-all", X, Y], [[tag, forall], [ids, X], [implication, Y]])`) and “share” the arguments between the DSL and the IR. This, however, is not possible in *core.logic* (see Section 3.1 for details).

```
1 (defn translato [dsl ir]
2   (fresh [... introduce variables below ...]
3     (conso operator args dsl)
4     (featurec ir {:tag ir-tag})
5     (db/rules ir-tag operator db-tags)
6     (conso _1 _2 db-tags)
7     (try-extract-mappo db-tags ir ir-pairs)
8     (match-vals-keys db-tags translated-args ir-pairs)
9     (conso [:tag ir-tag] ir-pairs ir-pairs-with-tag)
10    (maplisto translato args translated-args)
11    (pairs-mappo ir-pairs-with-tag ir)))
```

**Fig. 1.** Main Case of the Translation Relation

<sup>5</sup> The full code base can be found at [https://github.com/pkoerner/lisb/tree/feature/core-logic-translation/src/lisb/translation/core\\_logic\\_translation](https://github.com/pkoerner/lisb/tree/feature/core-logic-translation/src/lisb/translation/core_logic_translation).

<sup>6</sup> To the best of our knowledge, one cannot express the example using SWI’s `dicts` because the equivalent of `featurec` (cf. Section 3.2) requires ground arguments.

```

rule(forall, "for-all", [ids, implication]). % data base, cf. Sec. 3.1
make_tuple(X,Y,[X,Y]). % helper predicate for a concise maplist

translato([Operator|Args], IR) :-
  select(['tag', IR_Tag], IR, IR_Pairs),
  rule(IR_Tag, Operator, DB_Tags),
  maplist(make_tuple, DB_Tags, TArgs, IR_Pairs),
  maplist(translato, Args, TArgs).
translato(X,X) :- atomic(X). % example of direct translation of primitives

% example call --- can be called in both 'directions'
?- translato(["for-all", a,b], X), !, translato(Y,X).
X = [[tag, forall], [ids, a], [implication, b]],
Y = ["for-all", a, b] .

```

Fig. 2. Prolog Example Approximately Mirroring Fig. 1

### 3.1 Data Base and Variadic Arguments

*core.logic* allows executing queries in the context of data bases; yet, in contrast to Prolog, once a variable in the knowledge base is bound, it remains bound to that value for the entire query (in particular, another lookup yields the same value). Thus, we organise only the static components of the translation rules as a ternary data base relation. The first item is the value in the IR map associated under the key `:tag`, identifying the operator. Second is the symbol that is the operator in the DSL. The third entry is a vector of keywords giving the syntactic ordering of the arguments. For example:

```

[:for-all for-all [:ids :implication]]
[:add + [[:nums]]]
[:any any [:ids :pred [:subs]]]
[:nat-set nat-set []]

```

The `(for-all ...)` form expects two arguments, first a vector of identifiers, and second a logical predicate that must be fulfilled. The addition allows an arbitrary number of arguments: The extra vector around `:nums` is syntax meaning *all* following arguments will be stored under this key. Both concepts come together for the DSL call `(any ...)`, where first comes a vector of identifiers, second a predicate constraining the identifiers, and third an arbitrary number of variable substitutions. Last, the constant `nat-set` has no argument and should not be a call but a symbol in the DSL.

*core.logic* allows specifying which arguments should be used to index a relation in the data base (in contrast to most Prolog systems which use the functor of the first argument for indexing). In this instance, we chose both the first and the second argument to avoid a linear search, allowing efficient lookup (line 5) both when the first argument is known (translating the IR to the DSL) and when the second argument is known (translating the DSL to the IR).

### 3.2 Locating Operator and Tag

Before we can perform the data base lookup, we need to destruct the DSL into operator and arguments (line 3). If the DSL is given, we can obtain the keys for the IR map. Given the IR, we need to find the corresponding operator and to know which keys contain the arguments (in which order).

Unfortunately, support for the map data structure is severely limited (ultimately, due to the implementation in Clojure as an immutable version of hash tries [2]). The *core.logic* library offers a `featurec`-function that sets up a constraint that at least a given key-value pair is present in the map. While one can use a logical variable as a value, variable propagation does not work properly, and the resulting map may still have an open variable with an additional constraint. However, it is sufficient to extract the value under a given key (or keep it as a logical variable, line 4), so that the data base lookup works bi-directionally.

### 3.3 Generation of Maps

Knowing both the DSL operator and the IR tag, the next step is to recursively translate the arguments (line 10).

*If the IR is given*, we need to extract the values (line 7), order them (defined by the data base entry, line 8), and recursively translate them back into the DSL (line 10). In order to extract the values, we wrote a relation that iterates over a sequence of keys and yields the corresponding key-value pairs by repeatedly projecting a function that performs the map lookup (line 7).

However, *if the DSL is given*, we need to translate the arguments *before* we are able to construct a map (as `featurec` calls are uni-directional and no alternative exists — line 7 becomes a no-op if the map is not bound). Thus, we collect the key-value tuples in a list (line 8 and 9), use `project` to obtain the value of this logical variable and non-relationally unify the IR with a new map created from these tuples (line 11).

## 4 Discussion

Having achieved a bi-directional translation, in the following, we want to discuss our experience during development.

### 4.1 Goal Achievement

First, we want to emphasise that almost everything we aimed for is possible! However, in most cases, it comes with a caveat: In particular,

- we can transform DSL code to the IR and back using the same code;
- the translation rules are represented compactly as data base facts (albeit less readable than we hoped), see Section 4.2;
- changes that concern *all* operators can be made in *one* location (but correct changes are harder than before, see Section 4.4);

- the implementation *technically* works for large inputs (although the performance is an issue, see Section 4.4).

Due to our unfamiliarity with the search strategy, only consistently providing error messages that highlight unsupported constructs in either the DSL or the IR instead of returning no solution was not achieved. Yet, in most tested scenarios that did not provide *both* the DSL and the IR, it seems to work as intended.

## 4.2 Code Size

The resulting implementation has about 170 data base entries in addition to about 150 LoC (lines of code) for the *core.logic* relations. The main relations consist of 15 clauses, of which six are for special cases and two attempt to output useful error messages. This is a significant improvement over the old name space spanning over 1400 LoC for *one* translation.

## 4.3 Test Strategy

To validate completeness and correctness of the new translation and to locate bugs, we used a combination of tests: First, example-based tests that check whether the translation yields the correct result for small snippets. Second, tests that check whether executing works bi-directionally, i.e., the re-translation yields the input again. Third, we used `test.check`, a Clojure implementation of QuickCheck [8] to generate DSL code (which may contain type errors that are not checked at this stage). The property used for testing was that the new translation yields the same IR as the reference implementation.

The property-based approach quickly uncovered nodes missing in the data base, in particular some constants and aliases. It also showed that our transformation from sets to sequences did not work for the empty set: Instead of the empty list, `nil` was returned, which was not handled by our implementation of `maplisto`.

## 4.4 Performance

As *core.logic* can be regarded as an interpreter, we were prepared to accept slower performance than our (not heavily optimised) reference implementation<sup>7</sup>. However, for some generated input, the translation was surprisingly slow (more than a minute runtime even if the code snippet only spanned over three lines).

We first assumed that — due to our unfamiliarity with the search strategy — too many alternatives were explored and some `conde` should be replaced by a version that hinders backtracking. Our investigation led to an innocent

**Table 1.** Runtime of (`nesto n 1`) calls.

<i>n</i>	runtime (msecs)
17	94.829496
18	186.944613
19	372.794756
20	751.533654
21	1534.125473
22	3074.324231
23	6066.818982
24	12140.838974

<sup>7</sup> Compared to native code, often at least a 10× slow-down is mentioned [4].



minimal relation (`nesto n l`) that relates a number `n` with a list `l` that is nested `n` times, e.g., (`nesto 0 []`) and (`nesto 3 [[[[[]]]]`) hold true:

```
(defnu nesto [n l]
  ([0 []]) ;; pattern matching of arguments - base case
  ([n [res]] ;; recursive case - nest the result
   (fresh [nn]
    (is nn n dec) ;; decrease number
    (nesto nn res))))

;; Prolog version
nesto(0, []).
nesto(N, [Res]) :- NN is N - 1, nesto(NN, Res).
```

Indeed, querying a single solution is exponential in  $n$  in *core.logic* (cf. Table 1) (but is linear in Prolog). The same behaviour does not only hold true for nested sequences (which is our DSL), but also for nested maps (which is our IR). In fact, an almost nine-year-old issue<sup>8</sup> identifies that even *unifying* with a deeply nested data structure performs slowly. The underlying issue is a non-optimal implementation of the unification algorithm that for each level of nesting tries to unify the entire (sub-)structure again, even though child nodes have already been visited.

#### 4.5 Required Expertise

The resulting relation and required helper relations are definitely on the short side. We want to stress that the endeavour is not even *hard* per se. Yet, we noticed that the entry barrier is very high.

An issue is that innocent-looking changes or attempts to simplify the code easily can introduce bugs. As an example, we used calls exactly as the `conso` in line 6 in Fig. 1 to generate the correct data type, i.e., lists instead of vectors. This instance, however, is *not* such a trick; In fact, it is necessary to distinguish operators from constants. An accidental removal of the call did not obviously break anything; in fact, it just added solutions (e.g., additionally to `nat-set` the wrong DSL call (`nat-set`) was supported as well). However, there was an — again innocent-seeming — symptom in the tests: Translating the call (`contains? nat-set 42`) to the IR and back suddenly did not yield itself anymore but instead its alias (`member? 42 nat-set`). While we do not entirely understand the reasons, the additional solutions somehow influenced the search order, so that the *first* solution changed as well.

*Standard Library* The underlying *cause* is that there is no real standard library of relations, neither built-in nor provided by the community. The list of built-ins in Section 2.1 is almost exhaustive! Neither most typical Prolog predicates nor most Clojure functions are available. If there was a `nonempty`-relation, the reason for the call in line 6 would have been clear.

<sup>8</sup> <https://clojure.atlassian.net/browse/LOGIC-177>

*Documentation* The documentation of *core.logic* is very concise. As a metric, there are 191 public vars in the core namespace, of which 120 are not documented. It is hard to guess which parts of the library could be relevant. The Wiki attached to its GitHub repository offers some information, but is not very organised. Worse, for both docstrings and the wiki, there are open issues in the bugtracker that some of the already little information is plain *wrong!*

*Debugging* Debugging is severely hindered in *core.logic*, as internally, everything seems to be an anonymous closure. The function `trace-lvars` allows outputting the current variable bindings. Yet, due to the search strategy (and laziness of the sequence of results), it does not offer any insights which call actually is relevant for the result. There seems to be no further functionality.

*Readability* While it is a fairly subjective measurement, the syntax of *core.logic* seems unfamiliar both from a Clojure and from a Prolog view. It might be that pattern matching is only available as part of the `matche` expression (and its variants), so that even list structures are lost to the reader’s eye behind variable names and calls to `conso`.

*Data Structure Support* The integration of *core.logic* with the basic map data structure is, unfortunately, not good. As long as maps are fully instantiated, it is reasonable to work with them. However, if maps are constructed on the fly, it seemingly becomes impossible to do it purely relational. Even simple constraints on maps, that require a key-value pair as a feature in a map with a bound key and variable value do not propagate bound values into the solution. The map remains unbound with an open feature constraint. There also seems to be no way to obtain a minimal, labeled value. As an example, in the following call, `m` will remain unbound unless explicitly unified with a fully instantiated map:

```
(featurec m {:foo 42})
```

Our solution ultimately falls back to basic lists of key-value tuples. It is unfortunate that it is required to ultimately mimic Prolog-style predicates where maps usually are not supported at all. The same goes for the set data structure, where even enabling basic support feels much like a dirty hack.

## 4.6 Related Work and Alternatives

Overall, there is little documented use of the *core.logic* library: most notable are Typed Clojure [3], which adds a static type system to Clojure; and FunnyQT, a library to query and transform models [10, Ch. 33–37]. It offers a DSL that allows users to express transformations between representations of the same model, with additional conditions on the relation that must be fulfilled. The *core.logic* library was also used in a mini case study in linguistic computing [18] to encode automata as a logical relation.

Nogatz et al. [17] [16, Ch. 9] implemented a tool in Prolog to both infer coding style rules of Prolog predicates and to pretty print the AST in accordance with

such rules. Many of the supplied predicates can be called bi-directionally, i.e., they are agnostic to their call mode. This ultimately yields a tool that provides both a parser and a pretty printer using the same Prolog code. The employed ideas are largely similar to the ones used in this work. The main difference lies that the Prolog tool utilises delays (e.g., to avoid instantiation errors on string operations) while we have to resort to using specialised helper relations based on the argument that is instantiated.

*Alternatives* The *data base structure* presented in Section 3.1 was chosen somewhat arbitrarily. A Prolog-style data base with logical variables would have been preferred — however, once bound, the variables remain unified in *core.logic*. A more verbose solution could have performed a term-copy after the look-up and unify the values with the copied variables. Alternatively, one could have designed a large match-clause that retains both the DSL sequential structure and the IR map structure as literals, such as:

```
(matche [dsl ir]
  ([[for-all ids impl]
    {:tag :for-all :ids ids, :implication impl'}]
   (translato impl impl'))
  ([[+ . args] {:tag :add, :nums args'}]
   (maplisto translato args args'))
  (['nat-set {:tag :nat-set}])
  ...)
```

The main advantage is that the awkward construction of maps is not necessary. Yet, changes afflicting all clauses, e.g., retaining meta-data, requires modifying all clauses or adding a wrapping relation that is mutually recursive with the main translating relation.

Another alternative would be to use a less mighty library such as meander<sup>9</sup>, a *term re-writing system*. While one can easily express translation rules, it seems that only uni-directional transformations are possible. Using a data-base like approach here could allow us to simply generate both directions.

Further, one could just directly use a *Prolog* — preferably one that runs on the JVM (e.g., 2P-KT / tuProlog [7]) for ease of integration with other features of *lisb*. However, further transformation between supported Prolog data structures and Clojure data is, again, required, similar to outlined issues with maps in *core.logic*.

## 5 Conclusion

Overall, miniKanren and the *core.logic* implementation in particular have the potential to be tools that are very useful and suitable for such a task like bi-directional transformations between arbitrary data. Yet, before we can consider

<sup>9</sup> <https://github.com/noprompt/meander>

using it in practice, the following points would need to be addressed (ranked in the order of our subjective severity):

- Most pressing is the performance issue in the unification of nested data structures (cf. Section 4.4). This is, ultimately, the blocker preventing the use of the translation we developed. Even though there are no hard performance constraints (we would be happy to trade one or two orders of magnitude of runtime for code that is more maintainable), the exponential runtime (regarding the depth of nested sub-expressions) absolutely hinders adaptation of the *core.logic* implementation; After all, the nesting level can quickly become deep in complex or generated mathematical constraints.
- The debugging and tracing capabilities need to be improved. Currently, the search strategy is hard to comprehend, leading to information that is seemingly randomly interleaved. It is almost impossible to locate bugs based on the tracing output. This directly impacts the maintainability of the code written in *core.logic* and would be a significant trade-off when considering using the new implementation.
- Clojure’s standard data structures, i.e. maps and sets, need better integration. Pragmatic limitations would be fine: it would suffice if one could generate a minimal map given the `featurec` constraints, or use variables as values in maps. At its current state, the interface suggests that more features exist than are available.
- Even though there is a significant amount of information on the usage of *core.logic* offered in tutorials, the documentation still needs to be improved. For example, it is hard to understand the semantics of the `conda` and `condu` variants that implement versions of a cut operator. Paired with the incomprehensive search strategy, it is also hard to learn it by small experiments.

This example aligns with our overall experience regarding the declarative paradigm: techniques and tools are well-known and available. However, the shift to bring it to *mainstream* programmers — here, with access to the entire Java eco-system — did still not occur: Tools often are specialised for their own input language and lack a good integration with traditional programming languages. Issues which are software engineering problems, like performance bugs or memory leaks, often remain unresolved. The reasons for obtaining a result are sometimes hard to follow (think Prolog’s “no”), and debugging tools sometimes are not sufficient. Further, in many cases, documentation is lacking.

Ultimately, the implementation work might only have been possible because the author has a background in programming with Prolog. Regardless, the experience remained overly cumbersome and, ultimately, disappointing. We hope that declarative techniques may eventually be more widely adapted. However, they first need to grow more mature and accessible to non-academic programmers.

**Acknowledgments.** The author thanks Mounira Kassous for the prototype of the implementation in *core.logic* and Henrik Hinzmann for implementing the generators for testing.

## References

1. Abrial, J.R.: *The B-Book: Assigning Programs to Meanings*. Cambridge University Press (1996)
2. Bagwell, P.: Ideal hash trees. *Es Grands Champs* **1195** (2001)
3. Bonnaire-Sergeant, A.: *Typed Clojure in Theory and Practice*. Ph.D. thesis, Indiana University (2019)
4. Brunthaler, S.: Virtual-machine abstraction and optimization techniques. *Electronic Notes in Theoretical Computer Science* **253**(5), 3–14 (2009). <https://doi.org/10.1016/j.entcs.2009.11.011>
5. Butler, M., Körner, P., Krings, S., Lecomte, T., Leuschel, M., Mejia, L.F., Voisin, L.: The First Twenty-Five Years of Industrial Use of the B-Method. In: *Proceedings FMICS (International Conference on Formal Methods for Industrial Critical Systems)*. *Lecture Notes in Computer Science*, vol. 12327, pp. 189–209. Springer (2020)
6. Byrd, W.E.: *Relational programming in miniKanren: techniques, applications, and implementations*. Ph.D. thesis, Indiana University (2009)
7. Ciatto, G., Calegari, R., Omicini, A.: 2P-KT: A logic-based ecosystem for symbolic AI. *SoftwareX* **16**, 100817:1–100817:7 (2021). <https://doi.org/10.1016/j.softx.2021.100817>
8. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: *Proceedings ICFP (International Conference on Functional Programming)*. pp. 268–279. ACM (2000)
9. Hickey, R.: A History of Clojure. In: *Proceedings HOPL (History of Programming Languages)*. pp. 1–46. ACM (2020)
10. Horn, T.: *A Functional, Comprehensive and Extensible Multi- Platform Querying and Transformation Approach*. Ph.D. thesis, University of Koblenz-Landau (2016)
11. Körner, P., Mager, F.: An Embedding of B in Clojure. In: *Companion Proceedings MODELS (International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings)*. p. 598–606. ACM (2022)
12. Körner, P., Bendisposto, J., Dunkelau, J., Krings, S., Leuschel, M.: Integrating formal specifications into applications: the ProB Java API. *Formal Methods in System Design* **57**, 160–187 (2020)
13. Körner, P., Leuschel, M., Barbosa, J., Costa, V.S., Dahl, V., Hermenegildo, M.V., Morales, J.F., Wielemaker, J., Diaz, D., Abreu, S., Ciatto, G.: Fifty Years of Prolog and Beyond. *Theory and Practice of Logic Programming* pp. 1–83 (2022)
14. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. *Software Tools for Technology Transfer* **10**(2), 185–203 (2008)
15. Miller, A., Halloway, S., Bedra, A.: *Programming Clojure*. Pragmatic Bookshelf, 3rd edn. (2018)
16. Nogatz, F.: *Defining and Implementing Domain-Specific Languages with Prolog*. Ph.D. thesis, Universität Würzburg (2023)
17. Nogatz, F., Seipel, D., Abreu, S.: Definite Clause Grammars with Parse Trees: Extension for Prolog. In: *Proceedings SLATE (Symposium on Languages, Applications and Technologies)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2019)
18. Varjú, Z., Littauer, R., Ernis, P.: Using clojure in linguistic computing. In: *Proceedings ELS (European Lisp Symposium)* (2012)