pymicrolog – Interactive Logic Programming in Python

Mario Wenzel[1]

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany
mario.wenzel@informatik.uni-halle.de

Abstract. pymicrolog is an embedded DSL for the popular programming language Python implementing Microlog. Microlog is a logic programming language based on Datalog with an explicit call-convention that allows for interaction with imperative library code.
This paper describes the syntactic and semantic differences between Microlog and pymicrolog. A few example applications are shown to underscore the viability of Microlog as a general purpose programming language for interactive applications.

Keywords: Logic Programming · Datalog · Python.

1 Introduction

Games programming and robotics are two of the main draws in computer science outreach. Interactive applications have been used to make computer science exciting and graspable for a very long time [9]. While mathematical operations on a blackboard are usually free of side-effects, with games the interactive parts of a computation are the important parts of a program. Similarly, the current game state (e. g., game board, map) are mostly or completely visible and this makes games inherently stateful.

The programming languages and paradigms that are used in school-based and extracurricular courses are limited. Programming courses and computer science outreach programs based on programming games and robots mostly devoid of other programming paradigm than imperative programming.

But the declarative paradigm is arguably more useful for the general population than the imperative one. Spreadsheet software, for example, is at the core of many business operations and the importance and pervasiveness of Excel is obvious. Therefore it is not a surprise that an introduction to spreadsheet software is part of most school curricula or introductory adult computer literacy courses. The computational model of Excel and similar spreadsheet software is not an imperative one.

Query languages, as another example, be that SQL, SPARQL, Datalog, or others, are useful tools for information retrieval. In schools, library access is sometimes taught using a domain specific query language employed by the local

or institutional library. More broadly, "how to google" is generally seen as an important skill. SQL is still seen as a core language in a business intelligence toolkit [5]. Again, the computational model of query languages is usually not an imperative one. We value query languages because quite the opposite is true: we tell the computer system what to retrieve, and not how to retrieve it.

Besides a recent experiment in Argentina [3], where primary school children were tasked to find a thief using a Prolog based detective game, current experiments in early computer science education and computer science outreach seem to cover logic and its application only sparsely. We do find robotics courses and game programming that use graphical imperative programming languages (like "Scratch" [7]) to encode behaviour which should foster "computational thinking" [6]. In terms of artificial intelligence (AI), the current trend seems to go away from explainable AI, and towards statistical methods. The question is, whether these skills will be, now and in the future, as broadly applicable as methods with stronger links to classical logic.

We would like to employ the same engaging topics, like robotics and games programming, and still keep the benefits of declarative logic programming:

- It is easier to teach and learn, as it keeps more closely to pure mathematics and the known semantics of the blackboard.
- It is more broadly applicable as interfaces to many interactive systems are not imperative in nature. And even if they are, the statefulness can be modeled explicitly.
- Programs can more easily be shown to be correct, as the program semantics is easier encoded in mathematical logic.

In this paper we present pymicrolog, an embedded Python DSL encoding most of the Microlog semantics. Microlog is a logic programming language based on Datalog with special relations that map to function calls of an imperative system. Facts deduced in these special relations lead to corresponding function calls. Resulting data from calls to external functions is fed back into the system as the initial input for the next state deduction. This leads to a coinductive model that allows for non-termination. And indeed we do not even allow for termination. We can interpret a Microlog program as a stream of function calls with their return values fed back into the system.

While Microlog is easily compiled as standalone microcontroller programs, it is a bit unwieldy to use within other interactive contexts, e. g., embedding it into an existing programming framework.

pymicrolog tries to bridge that gap, as the embedded Python DSL can easily be used to implement logic programs in existing Python-based imperative frameworks for user interaction, like the EV3dev Python bindings for LEGO EV3 or terminal control libraries like curses.

In Section 2 of this paper we recapitulate the Microlog language and explore some syntactic and semantic differences between Microlog and pymicrolog. Section 3 explores some example applications, proving the viability of pymicrolog for implementing interactive applications. Section 4 sees us draw conclusions and problems noted from the implementation of the example applications.

2 Microlog recap and pymicrolog differences

Microlog is based in Datalog with an explicit representation of time is modeled after the Dedalus$_0$ language [1]. Broadly, Microlog contains four kinds of rules:

- Initial facts `p(1, 2)@0.` are facts that are true at the start of the program for timestamp 0. In contrast to Datalog, where the initial database for the extensional relations is seen as "swappable", while the intensional relations defined by rules are seen as the "program", Microlog has no concept of a swappable extensional database. The extensional database are (first) the initial facts (`@0`) and subsequently the inductive facts passed into the future (`@next`).
- Deductive rules `p(X) :- q(X), #f(_, Y), X < Y.` that deduce additional facts for the current timestamp and correspond to Datalog rules for intensional relations.
- Inductive rules `p(X)@next :- !p_del(X), p(X).` deduce facts that are true for the next timestamp and are part of the extensional database for the next program run.
- IO rules `#f(X, ?)@next :- p(X).` deduce function calls (here for the library function `f` with one argument) and the result of the call is put into the `?` so that some fact `#f(X, R)` is part of the extensional database for the next timestamp.

Besides regular predicates, arbitrary quantifier-free formula of some chosen theory (like Presburger Arithmetic) with variables appearing in positive body literals are allowed. There are some differences between pymicrolog, the embedded Python DSL, and the concrete syntax for Microlog that is compiled to C++.

2.1 Syntax

Syntactically the Python DSL is based on operator overloading. Therefore we are limited to the operators in the Python language, and try to follow Python programming conventions where possible. These are some basic syntactic differences:

Microlog concrete syntax	pymicrolog
`@next`	`@NEXT`
`@0`	`@START`
`!`	`~`
`#`	n.a.
`?`	n.a.
`:-`	`<=`
`,`	`&`
`_`	`...`
`.`	`,`

2.2 Declaration

As in other python DSLs, all used symbols need to be declared beforehand. There are no syntactic limitations regarding the symbols (besides being Python identifiers and not keywords), but the convention of starting variables with upper

case letters and everything else with lower case letters works fine with the usual Python naming conventions.

- Relations need to be declared and assigned to a Python variable. But they are variadic and no type information is stored.
  Example: `marker = relation("marker")`
- IO predicates that refer to callable functions are declared with the callable object as its argument. As they are also assigned to Python variables, they may not start with an octothorpe. IO predicates, like relations, are variadic.
  Example (function object): `announce = call(print)`
  Example (lambda expr.): `get_input = call(lambda : stdscr.getch(0,0))`
  IO predicates may also be declared with a non-callable but hashable object as its argument, which allows the user to supply the callable at a later point.
  Example (non-callable): `announce = call("print")`
- Variables to be used in rules need to be specially created. Though they can be reused in multiple rules.
  Example (single variable): `L = variable("L")`
  Example (multiple variables): `C1, C2 = variables("C1", "C2")`
  Example (multiple variables unpacked): `C, R, P = variables(*"CRP")`
- There's no special theory attached to the pymicrolog semantics, and theory formula are just Boolean oracles from callables. The called function should be free of side-effects, but that is not enforced:
  Example (function object): `lt = oracle(operator.lt)`
  Example (lambda expr.): `lt = oracle(lambda x, y: x < y)`

2.3 Variadic Relations and Calls

In pymicrolog all relations are variadic, i. e., the number of arguments is not fixed by the name or type of the relation. When facts in storage are matched with literals in rule bodies and the number of arguments differ, then the "shorter" literal or fact is extended to the same number of arguments with the `None` singleton. Therefore `r(A, B, C)` (all arguments being initialized as variables) matches with `r(1, 2)` with the substitution `{A: 1, B: 2, C: None}`.

This behavior is useful, as a Python function without return value (i. e., missing return statement or missing return value) implicitly returns `None`. The convention is that functions with side-effects always return `None` and only side-effect free functions return a value. Though that is just convention. Some functions in the Python standard library for regular expressions return `None` in case of there not being a match, and otherwise a match position or match object. Using the `return a,b` syntax, returning multiple values as a tuple is encouraged and also used quite often in the standard library.

A fact that results from an IO call has as its arguments the input arguments and then all return values (unpacked, if necessary). Our `None`-handling now allows us to easily work with the cases where a function returns `None` because it has side-effects and does not return anything, or because it deliberately returns `None` because that is the result of a search, for example. Given the following declarations

```
from pymicrolog import call, NEXT
from operator import add
fn_add = call(add)
fn_part = call(str.partition)
fn_announce = call(print)
```

the result fact on calling would be:

- `fn_add(5, 7)@NEXT` would result in `add(5, 7, 12)`
- `fn_part("Hello", "ll")@NEXT` would result in `str.partition("Hello", "ll", "He", "ll", "o")`
- `fn_announce("Hello")@NEXT` would result in `print("Hello",)` with the side-effect of "Hello" being printed to the command line.

This also works the other way around. As functions are variadic, they may be called with any number of arguments and we just pass all the input arguments from the corresponding call fact to the function. So `fn_announce("Hello", N)@NEXT <= user(N)`, with `N` declared as a variable works nicely and results in `print("Hello", name)` for each `name` in the `user` relation with the corresponding side-effect. `fn_announce("Hello", A, "from", B)@NEXT <= user(A) & user(B)`, deduces `print("Hello", a, "from", b)` for each pair of users `a,b`.

2.4 Python Data Model

It is also useful to note that we often can avoid `lambda` functions in `call` declarations as, in general, the Python function call `o.fn(arg)` is equivalent to `type(o).fn(o, arg)`. All functions attached to a class are static and are implicitly passed the calling object as its first argument if it was an object and not a class call. Usually we would call `"Hello".partition("ll")` instead of `str.partition("Hello", "ll")`, but they are equivalent. As functions are objects, they can be passed to the call-relation constructor.

This extends to oracles, where we can also use arbitrary functions attached to classes, like `oracle(str.isdigit)`, to check whether for the given string all characters in the string are digits and there is at least one character in the string.

Operators also have a specific calling convention. The convention is very complex, but as a simplification it suffices to say that `a <op> b` translates to `a.__<opname>__(b)`, which in turn translates to `type(a).__<opname>__(a, b)`. And if the types of a and b differ and the first operation raises a specific exception, the runtime tries `type(b).__<opname>__(a, b)` instead. As an example `"Hel" + "lo"` ultimately translates to `str.__add__("Hel", "lo")`.

2.5 Storage

There are two different storage backends for the relations that offer a trade-off between speed and flexibility in terms of which types of values are allowed:

- The **memory** backend uses Python sets to store the model and tuples to store facts. Therefore all hashable Python types are allowed as a term or IO call return value. The memory backend is relatively slow, as the data access for pattern matching is only linear.
- The **SQLite** backend uses an in-memory SQLite database to store the model. All facts are stored in a single relation. Only types that are supported by SQLite are allowed as a term or IO call return value. The SQLite backend is comparatively fast, as fact storage is index-supported.

2.6 Static Analysis

Microlog specifically targets microcontrollers and embedded devices. By only using oracle functions with known interpretations (corresponding to some user-chosen theory) and using this theory in an SMT-solver, the Microlog compiler analyzes Microlog programs and tries to prove that it uses finite memory and therefore has a finite state space. If the state space is finite, alternative compilation techniques, into a finite state machine, for example, are possible.

Additionally, through symbolic execution techniques the program may be minimized by removing rules and relations that are never used in a program run.

pymicrolog has no facilities for static analysis and only implements the default execution technique. pymicrolog programs are still checked for syntactic stratification, as this is a necessary precondition for the deduction algorithm to work, but otherwise there is no static analysis implemented.

2.7 Program Object and Main Loop

Regular Microlog is compiled into a device-specific template that includes the main loop. The Python DSL is to be embedded in regular Python programs.

The `Program` constructor has the signature `Program(rules, name=None, fnmapping=dict())` with the following arguments:

- An iterable of rules that the program is composed of.
- The name of the program for the purpose of string representation.
- A dictionary of mappings from arbitrary hashable values (the expected usage are strings) to callables, allowing oracles and call relations to be declared with an "name" that is replaced with an actual callable when the program is executed. This allows us to reuse programs and just exchange the callables without having to change the declarations.

3 Example Applications

In this section we look at a few example applications to get a feel for the viability of pymicrolog for programming interactive applications. Other example applications, like an edge follower implementation for the LEGO EV3 robot, is available in project repository.

3.1 Connect Four

It is possible to implement the two-player version of the popular "Connect Four" in pymicrolog using the `curses` module, which is an interface for the `curses` library for portable advanced terminal handling. Besides the initialisation and finalisation of the curses terminal, we can make do without imperative code. We have already noted in [10] that a 1-player version against the computer is difficult to implement. While it is possible to implement an AI opponent that exhaustively "looks" some n moves ahead and chooses one that is not losing, there is no random choice and there is no way to weigh board positions.

We start by importing our libraries and initializing the terminal. We register an exit call that waits for another key-press and then resets the terminal.

```
from pymicrolog import *
import curses, sys
stdscr = curses.initscr()
curses.noecho()
curses.cbreak()
stdscr.keypad(True)

def _exit(code):
    curses.nocbreak()
    stdscr.keypad(False)
    curses.echo()
    stdscr.getch()
    curses.endwin()
    sys.exit(code)
exit = call(_exit)
```

We also define some general calls to print single characters (in the board's coordinate system) and longer messages to the screen.

```
print_marker = call(lambda x, y, ch: stdscr.addch(10-y, x, str(ch)))
announce = call(lambda *a: stdscr.addstr(12,0, ' '.join(map(str,a))))
get_input = call(lambda : stdscr.getch(0,0))
refresh = call(stdscr.refresh)
```

The main difficulty for such a game is correctly scheduling the input and output with the `curses` library. *After* setting all the screen characters we need to refresh the screen. As the order of the calls within the same state is non-deterministic, we need one state where all the characters are written, and one where the screen is refreshed. The same is true for the phase where we receive the user input. Therefore we create a 4-phase state machine where in the "inbetween states" we refresh the screen. We also schedule the game to exit once a winner has been determined.

```
setup, phase, winner = relation("setup"), relation("phase"),
↪  relation("winner")
phase_rules = [ setup()@START,
```

```
    phase("out")@NEXT <= setup(),
    phase("settle_out")@NEXT <= phase("out"),
    phase("in")@NEXT <= phase("settle_out"),
    phase("settle_in")@NEXT <= phase("in"),
    phase("out")@NEXT <= phase("settle_in"),
    get_input()@NEXT <= phase("in") & ~winner(...),
    refresh()@NEXT <= phase("settle_in"),
    refresh()@NEXT <= phase("settle_out"),
    exit(0)@NEXT <= phase("settle_in") & winner(...),]
```

We create a few static facts to describe the state-space of our actual game. There are two players, seven columns, six rows, and static relations for describing the topology of the spaces on the board. We also create a few variables for players, rows, and columns.

```
player, current_player = relation("player"), relation("current_player")
player_facts = [ player(1), player(2), current_player(1)@START ]

column, row = relation("column"), relation("row")
top_of, besides = relation("top_of"), relation("besides")
column_facts = [column(x) for x in range(7)]
row_facts = [row(y) for y in range(6)]
top_of_facts = [top_of(y+1, y) for y in range(5)]
besides_facts = [besides(x+1, x) for x in range(6)]

C, R, P, P2 = variables(*"CRP", "P2")
C1, C2, C3, C4, R1, R2, R3, R4 = variables("C1", "C2", "C3", "C4", "R1",
↪  "R2", "R3", "R4")
```

We need to create relations for the board state and and how it changes. The board is initialized with all 0. If a player drops a marker in a column, the column is either empty, then the marker is at the bottom, or it is not empty and it replaces the empty marker that is on top of an existing marker. This is a straightforward implementation of "gravity".

New markers overwrite old markers for the next state, otherwise the board is carried over. We also print the board in the out-phase, and print the current player's number above the board to indicate the column that is currently selected for dropping their marker. All other indicator positions are to be cleared.

```
marker = relation("marker") # board state
new_marker = relation("new_marker") # where the new marker by the player
↪  will be
desired_column = relation("desired_column") # the currently selected
↪  column where a player maybe wants to drop their marker
dropped = relation("dropped") # the column where the marker is actually
↪  dropped
new_desired_column = relation("new_desired_column") # the column that the
↪  player will have selected in the next state
marker_rules = [
```

```
49      marker(C, R, 0) <= setup() & row(R) & column(C), # init
50      new_marker(C, 0, P) <= dropped(P, C) & marker(C, 0, 0),
51      new_marker(C, R, P) <= dropped(P, C) & ~marker(C, 0, 0) & marker(C,
        ↪  R, 0) & top_of(R, R2) & ~marker(C, R2, 0),
52      marker(C, R, P)@NEXT <= marker(C, R, P) & ~new_marker(C, R, ...),
53      marker(C, R, P)@NEXT <= new_marker(C, R, P),
54      print_marker(C, R, P)@NEXT <= marker(C, R, P) & ~new_marker(C, R,
        ↪  ...) & phase("out"),
55      print_marker(C, R, P)@NEXT <= new_marker(C, R, P) & phase("out"),
56      print_marker(C, 8, P)@NEXT <= current_player(P) & desired_column(C) &
        ↪  phase("out"),
57      print_marker(C, 8, " ")@NEXT <= column(C) & current_player(P) &
        ↪  ~desired_column(C) & phase("out"),]
```

The rules on how to handle the user input and when to register a dropped marker are quite straightforward. We have a new selected drop column if the user pressed "left" or "right" on the keyboard and there is still a valid column in that direction. Otherwise, the selected column stays the same.

If "down" is pressed on the keyboard and there is still space in the selected column, we get a dropped marker there for the current player. Once a marker is dropped, the current player switches, otherwise it stays the same.

```
58  get_drop_rule = [
59      desired_column(0) <= setup(),
60      new_desired_column(C2) <= get_input(curses.KEY_RIGHT) &
        ↪  desired_column(C1) & besides(C2, C1),
61      new_desired_column(C2) <= get_input(curses.KEY_LEFT) &
        ↪  desired_column(C1) & besides(C1, C2),
62      desired_column(C)@NEXT <= desired_column(C) &
        ↪  ~new_desired_column(...),
63      desired_column(C)@NEXT <= new_desired_column(C),
64      dropped(P, C) <= get_input(curses.KEY_DOWN) & current_player(P) &
        ↪  desired_column(C) & marker(C, ..., 0),
65      current_player(P)@NEXT <= dropped(...,...) & player(P) &
        ↪  ~current_player(P),
66      current_player(P)@NEXT <= ~dropped(...,...) & current_player(P),]
```

The rules on how to win a game are easily understood, if a bit verbose. We need to create a rule for one of the four directions ($\swarrow\downarrow\rightarrow\searrow$) and deduce a winner if there are four connected markers for the same player.

```
67  winner_rules = [
68    winner(P) <= player(P) & marker(C1, R, P) & besides(C1, C2) &
      ↪  besides(C2, C3) & besides(C3, C4) & marker(C2, R, P) & marker(C3,
      ↪  R, P) & marker(C4, R, P),
69    winner(P) <= player(P) & marker(C, R1, P) & top_of(R1, R2) & top_of(R2,
      ↪  R3) & top_of(R3, R4) & marker(C, R2, P) & marker(C, R3, P) &
      ↪  marker(C, R4, P),
```

```
70   winner(P) <= player(P) & marker(C1, R1, P)  & top_of(R1, R2) &
     ↪  top_of(R2, R3) & top_of(R3, R4) & besides(C1, C2) & besides(C2, C3)
     ↪  & besides(C3, C4) & marker(C2, R2, P) & marker(C3, R3, P) &
     ↪  marker(C4, R4, P),
71   winner(P) <= player(P) & marker(C1, R1, P)  & top_of(R1, R2) &
     ↪  top_of(R2, R3) & top_of(R3, R4) & besides(C2, C1) & besides(C3, C2)
     ↪  & besides(C4, C3) & marker(C2, R2, P) & marker(C3, R3, P) &
     ↪  marker(C4, R4, P),
72   announce("Player", P, "has won")@NEXT <= winner(P),
73   winner(P)@NEXT <= winner(P),]
```

As we already coded the program exit into a call, we simply run the program with all rules defined before.

```
74 Program(player_facts + column_facts + row_facts + top_of_facts +
   ↪  besides_facts + marker_rules + get_drop_rule + phase_rules +
   ↪  winner_rules).run()
```

While scheduling the input and output, which is inherently imperative, is a bit difficult, the game rules have easily been implemented and we get a terminal application for the famous Connect Four game in about 80 lines of code. Other terminal-based implementations took about twice as many lines [8].

3.2 Stratify

Using Microlog it is possible to implement the stratification algorithm from [4] in order to stratify regular Datalog programs. Ceri et al. describe the algorithm as such:

1. Construct the graph $EDG^*(P)$ from the extended dependency graph $EDG(P)$ for the program P as follows: for each pair of vertices p, q in $EDG(P)$, if there is a path with a negative edge from p to q in $EDG(P)$, add a negative edge between p and q in $EDG^*(P)$, if it does not already exist.
2. $i := 1$
3. Identify the set K of all those vertices from $EDG^*(P)$ without an outgoing negative edge. Output this as stratum i and delete all vertices of K and corresponding edges from $EDG^*(P)$.
4. If $EDG^*(P)$ is not empty, increase i by one and go to 3, otherwise stop.

We need to import all the constructors and define some variables and relations. The relation for a stratum is printed and the next stratum number is returned. We could also implement some external function for the successor, or define some finite subset of the successor relation.

```
1 from pymicrolog import *
2 A, B, C, V, S, R = variables(*"ABCVSR")
3 rel, edg, edgs, del_edgs, reachable, current_stratum = relation("rel"),
  ↪  relation("edg"), relation("edg*"), relation("del:edg*"),
  ↪  relation("reachable"), relation("current_stratum")
```

```
def _stratum_out(stratum, relation):
  print(stratum, relation)
  return stratum + 1
stratum, stratum_out = relation("stratum"), call(_stratum_out)
```

We define a set of initial facts that is the input data. For example the extended dependency graph for the "railway" program from [4].

```
rules = [
edg("safely_connected", "connected", 0)@START,
edg("safely_connected", "existscutpoint", -1)@START,
# ...
rel(R) <= edg(R, ..., ...), rel(R) <= edg(..., R, ...)
```

To find out, whether a path with a negative edge between two nodes exists, we construct the transitive closure of the extended dependency graph. Though we also need to carry the reachability via a negative edge: if A reaches B via a negative edge, then any C reachable from B is also reachable via a negative edge. We also need to consider the symmetric case.

```
reachable(A, A, 0) <= edg(A, ..., ...),
reachable(A, A, 0) <= edg(..., A, ...),
reachable(A, B, V) <= edg(A, B, V),
reachable(A, C, 0) <= reachable(A, B, 0) & reachable(B, C, 0),
reachable(A, C, -1) <= reachable(A, B, ...) & reachable(B, C, -1),
reachable(A, C, -1) <= reachable(A, B, -1) & reachable(B, C, ...)
```

We construct $EDG^*(P)$ according to the definition. If two vertices are reachable via a negative edge in $EDG(P)$, they receive a negative edge in $EDG^*(P)$. Otherwise we take the non-negative edges. We also table this relation, i. e. we carry the facts that are not in a specific "delete this"-relation.

```
edgs(A, B, -1) <= reachable(A, B, -1),
edgs(A, B, 0) <= edg(A, B, 0) & ~reachable(A, B, -1),
edgs(A, B, V)@NEXT <= edgs(A, B, V) & ~del_edgs(A, B, ...),
```

Finally, we need to identify the relations for the current stratum (those without negative outgoing edges), print them, delete (i. e., not carry over) all corresponding edges for the relations that we have assigned a stratum, and carry over the remaining vertices and edges, so that we can continue with the next stratum.

```
current_stratum(1)@START,
current_stratum(V) <= stratum_out(..., ..., V),
stratum(S, R) <= rel(R) & ~edgs(R, ..., -1) & current_stratum(S), # set k
↪  of no outgoing ~edge
del_edgs(B, V, C) <= stratum(..., V) & edgs(B, V, C), # delete them
rel(V)@NEXT <= rel(V) & ~stratum(..., V),
stratum_out(S, V)@NEXT <= stratum(S, V),]
```

And the result of running the program we get the following stratification, which is the stratification s_a from [4]:

```
1 station
1 connected
1 circumvent
1 linked
2 cutpoint
2 existscutpoint
3 safely_connected
```

4 Conclusion

As we have seen from the introduction of the 4-phase execution model for the connect-4 game, it would have been useful to be able to order the different IO calls so that there is a deterministic ordering within a single state. This would have made it possible to schedule the screen refresh after the drawing, sparing us from doubling the number of phases. We probably could have implemented the game using a 2-phase execution model (input and output phase). A prioritization is already present in [2], one inspiration for Microlog's model of IO, but was ignored as mandatory sorting of the output was seen as an undue computational burden in the microcontroller use case. IO ordering within the same has to be revisited and maybe introduced as an optional language construct.

Otherwise it is quite clear that interactive programs, like games, and other algorithms can be implemented using Pymicrolog with an expressive declarative syntax. Logic programming, at least in this case, lends itself to encoding game rules that can also be executed. As previously noted, Microlog is limited in its expressivity so that AI opponents that weigh multiple board states to choose from can not be implemented. Whether it is easier to then implement the whole game in imperative Python or whether it is easier to implement a callable that the pymicrolog code uses to solicit AI moves needs to be investigated. Though not shown in this paper, implementing robot behaviours for LEGO EV3 is another interesting use case that is well supported by pymicrolog.

The library is available at https://github.com/maweki/pymicrolog and has no additional dependencies outside the Python standard library.

References

1. Alvaro, P., Marczak, W.R., Conway, N., Hellerstein, J.M., Maier, D., Sears, R.: Dedalus: Datalog in time and space. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A.J. (eds.) Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers. Lecture Notes in Computer Science, vol. 6702, pp. 262–281. Springer (2010). https://doi.org/10.1007/978-3-642-24206-9_16, https://doi.org/10.1007/978-3-642-24206-9_16

2. Basol, S., Erdem, O., Fink, M., Ianni, G.: HEX programs with action atoms. In: Hermenegildo, M.V., Schaub, T. (eds.) Technical Communications of the 26th International Conference on Logic Programming, ICLP 2010, July 16-19, 2010, Edinburgh, Scotland, UK. LIPIcs, vol. 7, pp. 24–33. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2010). https://doi.org/10.4230/LIPIcs.ICLP.2010.24, https://doi.org/10.4230/LIPIcs.ICLP.2010.24
3. Cecchi, L.A., Rodríguez, J.P., Dahl, V.: Logic programming at elementary school: Why, what and how should we teach logic programming to children? In: Warren, D.S., Dahl, V., Eiter, T., Hermenegildo, M.V., Kowalski, R.A., Rossi, F. (eds.) Prolog: The Next 50 Years, Lecture Notes in Computer Science, vol. 13900, pp. 131–143. Springer (2023). https://doi.org/10.1007/978-3-031-35254-6_11, https://doi.org/10.1007/978-3-031-35254-6_11
4. Ceri, S., Gottlob, G., Tanca, L.: Logic Programming and Databases. Surveys in computer science, Springer (1990), http://www.worldcat.org/oclc/20595273
5. da Costa, L.: Self-serve dashboards, https://briefer.cloud/blog/posts/self-serve-bi-myth/
6. Fagerlund, J., Häkkinen, P., Vesisenaho, M., Viiri, J.: Computational thinking in programming with scratch in primary schools: A systematic review. Comput. Appl. Eng. Educ. **29**(1), 12–28 (2021). https://doi.org/10.1002/cae.22255, https://doi.org/10.1002/cae.22255
7. Maloney, J.H., Resnick, M., Rusk, N., Silverman, B., Eastmond, E.: The scratch programming language and environment. ACM Trans. Comput. Educ. **10**(4), 16:1–16:15 (2010). https://doi.org/10.1145/1868358.1868363, https://doi.org/10.1145/1868358.1868363
8. Nieves, O.: Programming a connect-4 game on python, https://oscarnieves100.medium.com/programming-a-connect-4-game-on-python-f0e787a3a0cf
9. Papert, S.: Mindstorms: children, computers, and powerful ideas. Basic Books, New York (1980)
10. Wenzel, M.: Expressivity of the microlog language. In: 35th Workshop on Logic Programming (WLP 2021) (2021)