Declarative Programming for Microcontrollers - Datalog on Arduino

Mario Wenzel

MLU Halle-Wittenberg

mario.wenzel@informatik.uni-halle.de

March 26, 2021

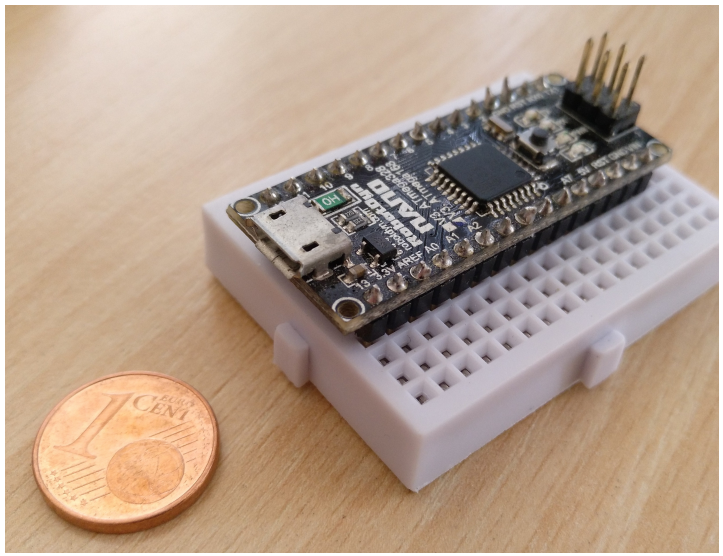
Presentation of ongoing PhD research

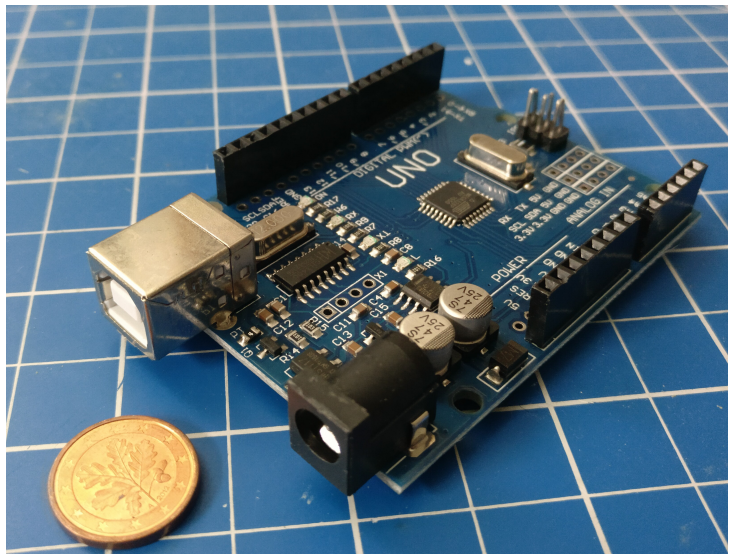
Declarative Logic Programming

- Declarative techniques allow us to write programs that relate closely to the specification (or even write compilable specifications).
- Declarative Logic Programming languages have mathematical precise semantics based on logic.
- LP is used in industrial applications and taught at university courses.

Microcontrollers

- Microcontrollers are ubiquitous: Internet of Things (IoT), home automation, model train control, sensing, other hardware control.
- Cheap (one target platform costs less than 1 EUR)
- Restricted Resources (2 KB SRAM, 32 KB program memory)
- Used in teaching in university and schools





Microcontroller Programming

- Not enough resources for operating system (programming "on the metal")
- Native C/C++ programming
  - Thousands of libraries are already availabe
- Few declarative approaches (MicroScheme, frp-arduino)
- No special LP framework
- No general purpose low-memory LP framework in C++
- Rule-based programming seems well-suited for microcontrollers

Microlog Language Overview

- Language based on Datalog with successor predicate
- All rules are range-restricted.
- No access to extensional Database
- All facts have an explicit timestamp
- Deductive Rules:
  - Normal Datalog deduction steps
  - $p(\mathcal{T}, X) \leftarrow q(\mathcal{T}, X, Y) \wedge p(\mathcal{T}, Y).$
  - `p(X) :- q(X, Y), p(Y).`
- Inductive Rules:
  - Govern how data flows through time
  - $p(\mathcal{T}', X) \leftarrow q(\mathcal{T}, X, Y) \wedge p(\mathcal{T}, Y) \wedge \text{succ}(\mathcal{T}, \mathcal{T}').$
  - `p(X)@next :- q(X, Y), p(Y).`
- Comparisons of arithmetic expressions in rule bodies allowed

Microlog Language Overview (IO)

- For a C-function `int digitalRead(uint8_t pin)` we introduce two predicates
  - Deducing call-fact (with set semantics) leads to function call *call_digitalRead*$(\mathcal{T}', P, ?)$, with ? for every output argument.
    short: `#digitalRead(P, ?)@next` as rule head
  - ret-facts are used to access return values (without side-effects):
    ret_digitalRead$(\mathcal{T}, P, R)$
    short: `#digitalRead(P, R)` in rule bodies.
- call-predicates are only allowed in next-rule heads, to preserve causality and finite state size.
- Whole statement blocks can be mapped to call/ret-predicates.

Leads to condition-action-rule:

$$\text{call\_f}(\mathcal{T}', X, ?) \leftarrow p(\mathcal{T}, X) \wedge \text{succ}(\mathcal{T}, \mathcal{T}').$$

short: `#f(X, ?)@next :- p(X).`

All allowed actions are taken, no negation in rule head, therefore no nondeterministic choice, no backtracking.

Statewise/Time-Stratified Deduction

The default Microlog runtime works by iteratively applying the program rules, discarding past states by omission of the timestamps.

- We rewrite the rules to standard Datalog program by
  - removing the timestamps
  - replacing @next by the prefix "next" (new predicates) and removing the succ-predicate
- Execute the Datalog Program
- Execute all function calls for the "call"-facts, creating instantiated return-facts (replacing the ?)
- Take all next-facts and return-facts and use them as seed facts (EDB) for the next iteration (removing the next-prefix)
- You can use any Datalog engine with very little plumbing code

State

- notion of deletion and updates
  (facts missing/changed from the "next" timestamp)
- notion of time (not everything happening at once)

Example:
```
table(X)@next :- add(X).
table(X)@next :- table(X), !delete(X).
```

Time	add	delete	table
101	add(1)		
102			table(1)
103	add(27)		table(1)
104			table(1), table(27)
...			table(1), table(27)
300		delete(1)	table(1), table(27)
301			table(27)

General Overview

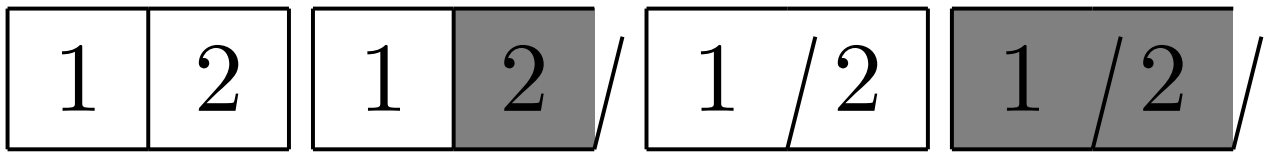
- Embedding into Datalog allows reuse of existing research, and engines
- Lack of dynamic database enables strong static analysis
- I/O can still be managed through strict call-convention, external calls and side-effects are never implicit
  - Value invention is restricted to IO
- Datalog with stratified negation is expressive enough for many use cases of embedded decision procedures

Compilation Approaches

- Usually: small datalog-runtime that executes deduction steps
  - In-memory database with deductive engine
  - Slow deduction
  - Very Strong memory restrictions
    - No generally proven memory bound
    - Usually no way to communicate OOM-Errors
- If finite memory proven: Compile Programs to Finite State Machines
  - Precomputation of possible interactions
  - Compilation to State Machine
  - Fixed and known memory requirements (by construction)

Example Application (Heating Control)

- If the window is open, heating should be off
  - Common application for home automation systems
- But rooms have doors and adjacent rooms may also have windows
- We want to shut off heating in all (transitively) adjacent rooms, if the doors are open
  - This can usually not be encoded in HA rule systems



Our example home: 2 rooms
1 and 2 are connected, 2 has a window
darkened rooms should have the heat turned off

Example Application Source (Heating Control)

```
% static example configuration (adjusted by user)
hasWindow(2).
adjacent(1, 2).
% gathering world state
#readWindow(R, ?)@next  :- hasWindow(R).
#readDoor(A, B, ?)@next :- adjacent(A, B).
% deduce model using transitive closure
windowOpen(R)   :- #readWindow(R, #open).
doorOpen(A, B) :- #readDoor(A, B, #open).
connected(A, B) :- doorOpen(A, B).
connected(B, A) :- doorOpen(A, B).
connected(A, C) :- connected(A, B),connected(B, C), A!=C.
% effects
#heatingOff(R)@next :- windowOpen(R).
#heatingOff(O)@next :- windowOpen(R), connected(R, O).
```

Abstract Deduction (General)

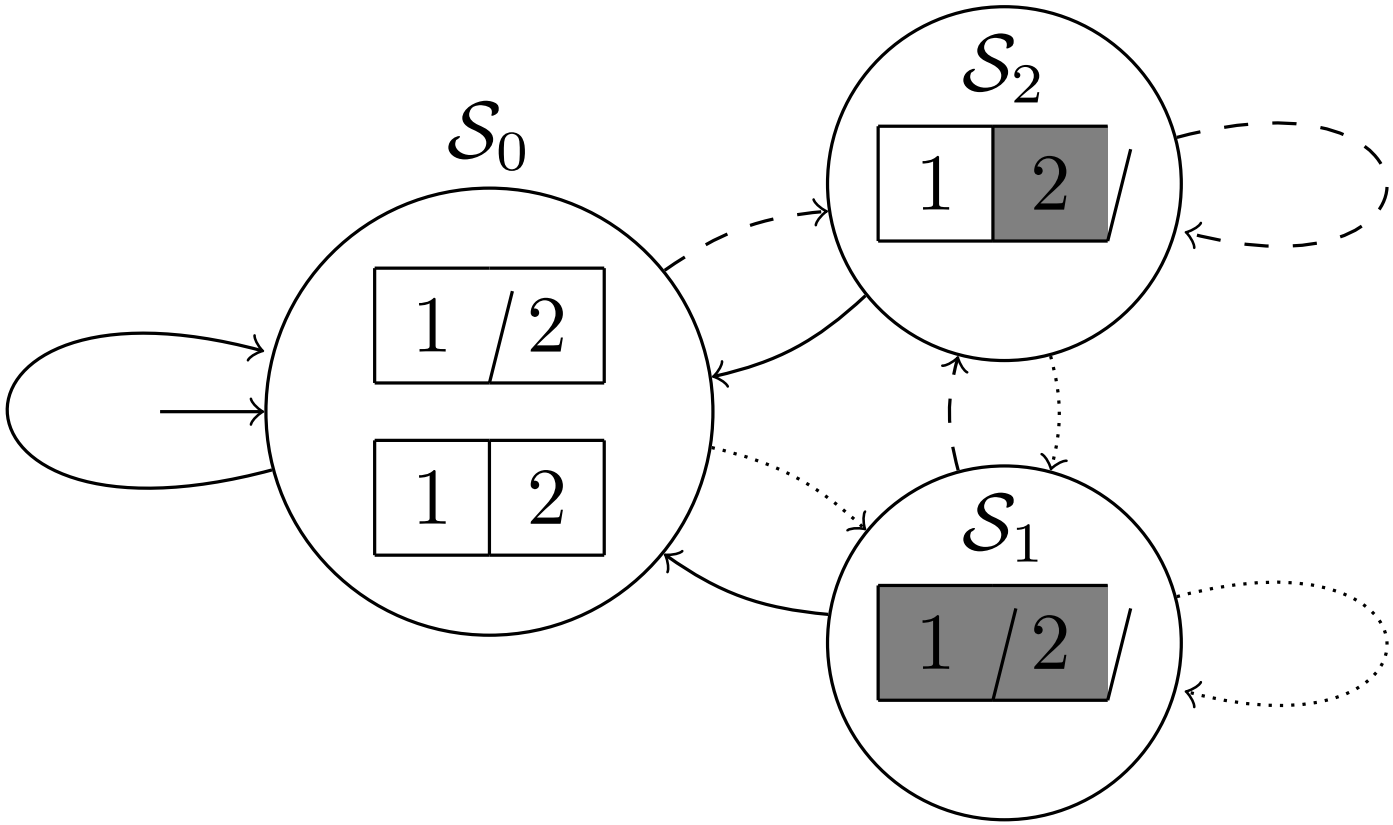
- Facts get attached conditions depending on IO return values
  - Example: `windowOpen(2)` if `#readWindow(2, #open)`)
- Static analysis requires an oracle for satisfiability in the theory of the arithmetic comparisons
  - the closer it matches the runtime, the better
  - an overapproximation is fine (at worst generates unreachable states)
- Conditional Facts with unsatisfiable conditions are removed
- Conditions with truth value depending on return values are checked at runtime
- Case distinction leads to state transition function

Abstract Deduction (Example)

- Replacing ? with variables, seed facts from previous state are
  - ret_readWindow$(2, V_1)$
  - ret_readDoor$(1, 2, V_2)$
  - Plus maybe ret_heatingOff(_)-facts (don't affect behaviour)
- case analysis leads to 3 different behaviours depending on the return values of the `readWindow` and `readDoor` calls.

$V_1 \neq$ #open	$V_1 =$ #open	
	$V_2 =$ #open	$V_2 \neq$ #open
readWindow(2, ?) readDoor(1, 2, ?)	readWindow(2, ?) readDoor(1, 2, ?) heatingOff(2) heatingOff(1)	readWindow(2, ?) readDoor(1, 2, ?) heatingOff(2)
$\mathcal{S}_0$	$\mathcal{S}_1$	$\mathcal{S}_2$

Finite State Machine



heatingOff$(R) \in \mathcal{S}$ R

ret_readWindow$(2, \#\text{open}) \notin \mathcal{E}_{next}$ $\longrightarrow$

ret_readWindow$(2, \#\text{open}) \in \mathcal{E}_{next}$ ·····›
ret_readDoor$(1, 2, \#\text{open}) \in \mathcal{E}_{next}$

ret_readWindow$(2, \#\text{open}) \in \mathcal{E}_{next}$ – –›
ret_readDoor$(1, 2, \#\text{open}) \notin \mathcal{E}_{next}$

Conclusion

- Microlog is a deductive logic programming language that is viable for microcontrollers and other interactive applications
  - No dynamic database, input/output is done explicitly
  - Strict call-convention
  - Approach is general (we also use it for LEGO EV3)
- The rule system is expressive
- If finite memory bounds can be shown, compilation to FSM is possible
  - We trade compilation time/static analysis for runtime and get a memory bound for free
  - We can check all states for additional constraints
- Related Ideas:
  - Logic Production Systems (Kowalski, Sadri)
  - Action Atoms/External Atoms in ASP (Eiter)

`https://dbs.informatik.uni-halle.de/microlog/`

Addendum: Divergent Program

```
read_sensor(?)@next.
table(X) :- read_sensor(X).
table(X)@next :- table(X).
```

- `table` could be optimized away as it never causes IO
  - We could add the rule `#out(X)@next :- table(X).`
- `read_sensor`'s return values are actually restricted by its type
  - We do not inspect any of the C-Code

Addendum: Conditional Facts

- formula of the form $p(t_1, \ldots, t_m) \leftarrow \varphi$
- each t_i is a constant or a parameter variable
  (memory locations for return values)
- φ is a consistent conjunction of atomic formulas $u \, \gamma \, u'$ with
  $\gamma \in \{=, \neq, <, \leq, \geq, >\}$
  (equality for unification, plus comparison predicates of runtime/oracle)
- u and u' are parameters or constants

We will do all possible deduction steps at compile time, delaying the evaluation of conditions until runtime.

Conditional facts have been previously used by Brass and Dix for semantic analysis of nonmonotonic negation.

Addendum: Conditional Facts (Example)

Replacing ? with fresh (pairwise distinct) variables instead of instantiations from actual calls:

$\text{ret\_readWindow}(2, V_1).$ $\text{ret\_readDoor}(1, 2, V_2).$	Generated Code: `V1 = readWindow(1);` `V2 = readDoor(1, 2);`

Consider the rule

$$\text{windowOpen}(R) \leftarrow \text{ret\_readWindow}(R, \#\text{open}).$$

applied to the fact

$$\text{ret\_readWindow}(2, V_1)$$

This leads to conditional fact

$$\text{windowOpen}(2) \leftarrow V_1 = \#\text{open}.$$

$(V_1 = \#\text{open})$ is the condition for successful unification

Applying rule $\text{call\_heatingOff}(R) \leftarrow \text{windowOpen}(R).$

leads to $\text{call\_heatingOff}(2) \leftarrow V_1 = \#\text{open}.$

Addendum: Calculating Successor States (Example)

In the example, the conditional successor state is:

$$\begin{aligned}
&\text{hasWindow}(2).\\
&\text{adjacent}(1,2).\\
&\text{call_readWindow}(2,?).\\
&\text{call_readDoor}(1,2,?).\\
&\text{ret_readWindow}(2,V_1).\\
&\text{ret_readDoor}(1,2,V_2).\\
&\text{windowOpen}(2) \leftarrow V_1 = \#\text{open}.\\
&\text{doorOpen}(1,2) \leftarrow V_2 = \#\text{open}.\\
&\text{connected}(1,2) \leftarrow V_2 = \#\text{open}.\\
&\text{connected}(2,1) \leftarrow V_2 = \#\text{open}.\\
&\text{call_heatingOff}(2) \leftarrow V_1 = \#\text{open}.\\
&\text{call_heatingOff}(1) \leftarrow V_1 = \#\text{open} \wedge V_2 = \#\text{open}.
\end{aligned}$$

Complete Example/Demo

```
.decl pressed
.decl setup
% IO Predicates
#pinIn(P: byte) = {pinMode(#P, INPUT);}
#pinOut(P: byte) = {pinMode(#P, OUTPUT);}
#digitalWrite(P: byte, Val: byte) = {digitalWrite(#P, #Val);}
#digitalRead(P: byte, Val: byte) = {int Val = digitalRead(#P);
% Input
#digitalRead(12, ?)@next.
pressed :- #digitalRead(12, #HIGH).
% Setup and Initialization
setup@0.
#pinIn(12)@next :- setup.
#pinOut(13)@next :- setup.
% Output
#digitalWrite(13, #HIGH)@next :- pressed.
#digitalWrite(13, #LOW)@next :- !pressed.
```