How To Write Meld Programs

Flavio Cruz

January 12, 2013

1 Introduction

Meld is a forward chaining logic programming language that uses linear logic as its foundation. Meld is similar to Datalog: a program consists of a database and a set of rules to derive new facts. Each fact is *predicate*/tuple pair and every element in the tuple must be a value.

Rules are composed of a *body* (the premisses) and a *head* (the conclusions). For a rule to be applicable, a subset of facts from the database must match the body of the rule. Once this happens, the head of the rule is instantiated and new facts are derived.

The database starts with a set of axioms. The database is updated by deriving new facts using the rules. Since Meld is based on linear logic, facts can be deleted from the database when applied to rules.

Meld is also a distributed language since facts are divided horizontally in different *nodes*. The first argument of each fact refers to the node where the fact is located. Rules are constrained in a way that facts used in the body must be located in the same node. However, facts in the head may refer to other nodes (this is where communication takes place).

2 Meld Program Structure

A Meld file consists of a header with predicate specifications and a set of rules that use those predicates. Constants and functions are also allowed and may be used in the rules.

2.1 Header

The header describes the facts that will be used in the program. Every predicate consists of a name, an argument type list and a set of modifiers.

Facts are then instantiations of predicates, created during execution, that consist of a pair of predicate and a tuple of values.

```
type [modifiers] name(type1, type2, ..., typeN).
```

Meld has a relatively rich set of types to choose from, namely:

int 32 bit signed integers.

float 64 bit doubles.

string a list of characters.

node represents a node address.

list int list of ints.

list float list of floats.

list node list of nodes.

boolean cannot be used in the predicate specification but it is an important type of Meld.

There are two types of modifiers: `linear` (predicate is linear, see next sections), `route` (predicate will be used for communication, see next sections) and/or `action` (fact represents an action).

2.2 Constants

Constants are expressions that can be used multiple times in the program as in other programming languages. They are represented by an identifier and an expression.

```
const identifier = expression.
```

Note that constants are computed only once (at the beginning of the program).

2.3 Expressions

Expressions can be used in rules as fact arguments or constraints and always evaluate to some value. The following lists all the possibilities:

⟨*expression*⟩ ::= ⟨*variable*⟩
| ⟨*arg*⟩
| ⟨*string*⟩
| ⟨*function-name*⟩ '(' ⟨*function-arguments*⟩ ')'
| ⟨*constant*⟩
| ⟨*node-address*⟩
| '(' ⟨*expression*⟩ ')'
| ⟨*number*⟩
| ⟨*world*⟩
| ⟨*expression*⟩ '+' ⟨*expression*⟩
| ⟨*expression*⟩ '-' ⟨*expression*⟩
| ⟨*expression*⟩ '*' ⟨*expression*⟩
| ⟨*expression*⟩ '/' ⟨*expression*⟩
| 'if' ⟨*cmp-expression*⟩ 'then' ⟨*expression*⟩ 'else' ⟨*expression*⟩ 'end'
| 'let' ⟨*variable*⟩ '=' ⟨*expression*⟩ 'in' ⟨*expression*⟩ 'end'
| '[]'
| '[' ⟨*expression*⟩ '|' ⟨*expression*⟩ ']'

⟨*world*⟩ ::= '@world'

⟨*arg*⟩ ::= '@arg' ⟨*number*⟩

⟨*cmp-expression*⟩ ::= '(' ⟨*cmp-expression*⟩ ')
| ⟨*cmp-expression*⟩ '||' ⟨*cmp-expression*⟩
| ⟨*expression*⟩ '=' ⟨*expression*⟩
| ⟨*expression*⟩ '⟨⟩' ⟨*expression*⟩
| ⟨*expression*⟩ '<' ⟨*expression*⟩
| ⟨*expression*⟩ '<=' ⟨*expression*⟩
| ⟨*expression*⟩ '>' ⟨*expression*⟩
| ⟨*expression*⟩ '>=' ⟨*expression*⟩

A few relevant notes:

node-address they start with `@` followed by a number and they refer to some node.

arg they refer to the arguments passed when the program started (think of `argv` in C) and have type `string`.

world it is an integer constant that returns the number of nodes in the program.

lists `[]` is the empty list and `[head | tail]` deconstructs or constructs a list.

cmp-expression this type of expression always returns a boolean value.

3 Rules & Axioms

Rules consist of a body and a rule. Axioms are rules without a body.

The basic syntax for rules is the following:

```
body -o head.
```

For axioms we simply have:

```
head.
```

We can also have headless rules, like this:

```
body -o 1.
```

3.1 Body

The body is composed of two main units:

Facts: They refer to facts from the database that must be matched. Each argument can be either a variable (starts with uppercase letter) or a simple constant. A constant enforces a rule constraint, while using variables allow the use of the corresponding fact argument elsewhere (in the head, other facts or constraints).

Constraints: Constraints are boolean expressions that must evaluate to true for the rule to match. They follow `cmp-expression` described above.

Variables always start with an uppercase letter and are used to refer to arguments of facts in the body. If a variable is used in the head, it must be also used in the body, since facts must be instantiated with literal values.

The following rule has a body with 3 facts and one constraint:

```
!heat(A, Temperature),
!desired-temperature(A, Desired),
Temperature > 30,
ac-turned-off(A)
   -o turn-ac-on(A, Desired).
```

The constraint `Temperature > 30` checks if the second argument of the `heat` fact is larger than 30.

3.2 Head

The head may contain facts (to be derived), comprehensions, aggregates or node instantiations. Constraints are not allowed, since they do not make sense here. Arguments of facts may have full expressions, as long as all variables inside the expression are visible in the body of the rule.

3.3 Functions

Small functions are also allowed in order to reduce repetitive code. They use the following syntax:

```
fun name(type1 V1, type2 V2, ..., typeN VN) : ret-type = expression.
```

Each function has a `name`, several arguments (from `V1` with type `type1` to `VN` with type `typeN`) and a return type `ret-type`. Function expression `expression` may use the variables declared as arguments. Note that all variables must start with an uppercase letter.

3.4 External Functions

The language provides several external functions that perform tasks that would be otherwise impossible to program. Here's the list of available functions:

randint(int X) : int Generates a random integer between 0 and `X`.

str2float(string S) : float Converts a string to a floating point.

str2int(string S) : int Converts a string to an integer.

truncate(float X, int Y) : float Truncates a float by the `Y`'th decimal place after the comma.

float2int(float X) : int Casts a float to an integer by truncating the float number.

int2str(int X) : string Converts an integer to a string.

float2str(float X): string Converts a float to a string.

str2intlist(string X): list int Converts a comma-separated list of integers to a list of integers.

3.5 Linearity

In Meld, we distinguish between *linear facts* and *persistent facts*. Persistent facts are never deleted and stay around *forever*. Linear facts stay around as long as they are not consumed. When a body uses a linear fact during the matching process, the fact is deleted from the database and can no longer be used.

Every time we refer to a persistent fact in a rule, we must use the *bang* ! symbol before the fact name. For example:

```
!edge(A, B), total(A, N) -o total(B, N + 1).
```

Note that `edge` is persistent, while `total` is linear. In this case `total(A, N)` is consumed and a new `total(B, N + 1)` is created at node B. The `edge` fact is not consumed since it is persistent.

3.6 Actions

Actions are linear facts that can only be derived but not consumed (they only show up in the head of rules). When an action is derived, something external happens and the fact is immediately consumed. These are some example actions:

- Change the color of a node.
- Change label of an edge.
- Prioritize computation.
- Halt computation.

Currently, the set of actions available is limited, but it is possible to extend it by changing the compiler source code and the virtual machine.

3.7 Selectors

When a rule is applied, the semantics of the language do not guarantee anything about the combination of facts that are used to match the rule. However, in some situations we want to force the rule to pick a certain fact before the others. This is specially important when we take into account linear facts. For example, we may want the rule to use the fact where the second argument has the smallest integer of all facts of that predicate. The basic syntax for selectors change the body of rules in this way:

```
[:sel Var | body ] -o head.
```

Currently, Meld supports two types of selectors:

min The fact with the minimum integer argument is selected first.

random Forces the runtime system to pick a fact at random. Although the operational semantics does not force any specific order, the runtime uses the most efficient order (implementation specific). This forces the implementation to actively randomize the available options.

When proving a rule with a selector, the first fact given the selector is picked and matching is attempted. If the matching fails, the next fact given the selector is tried, until the first fact succeeds. This is very useful since we can pick the best fact given the rule constraints.

The following code illustrates a common use case for selectors. We pick the minimum weight edge out of all edges, consume `send-message` and derive `message` on the best neighbor node.

```
[:min W | !edge(A, B, W), send-message(A, Content)]
      -o message(B, Content).
```

3.8 Comprehensions

In a nutshell, comprehensions are a mechanism that enables the repeated application of a linear rule. Although linear rules can only be applied once, there are certain situations where we want to apply a rule several times and then get rid of it. An obvious use case is scattering information across the neighborhood of a node. The syntax for comprehensions is the following:

```
main-body -o {variable-list | inner-body | inner-head}.
```

Note that comprehensions can only be used in the head of rules. `variable-list` is the list of variables introduced in `inner-body` that do not appear outside of the comprehension. Once `main-body` matches, `inner-body` is executed and if successful, `inner-head` is instantiated to derive new facts. The comprehension is continuously applied as long as `inner-body` matches.

A simple example would be the following:

```
fact(A, N) -o {B | !edge(A, B), limited(A, B) | fact(B, N)}.
```

For each `edge` that has a corresponding `limited` fact, we derive `fact` in node `B`. Note that `limited` is consumed during the application of the comprehension.

3.9 Aggregates

Aggregates allow the programmer to collect and aggregate information about several facts into a single fact. For example, one may want to sum the second argument of the set of linear facts into a new fact, that represents the sum of such values. Like comprehensions, aggregates can only be declared in the head of a rule and also have a list of variables that only appear inside the aggregate. The syntax template is the following:

```
body
   -o [operation => Result | variable-list | agg-body | result], head.
```

The variable `Result` will appear inside `agg-body` somewhere and represents the variable (from some argument) that we want to apply the aggregate to. This variable will also appear inside `result` in order to get the result as a fact. Operationally, `agg-body` is matched against the database using all the possible combinations, at each combination we add the current value of `Result` to a list. This aggregate list is then finally used to compute the result of the aggregate using `operation` and `result` is instantiated where variable `Result` refers to the aggregate value computed before.

Several aggregate operations are available, namely:

sum Sums all integers or floats.

min Selects the minimum integer or float.

count Counts the number of times `agg-body` has matched.

collect Collects every value into a list.

In the following example, we match every `distance` fact to determine the minimum value of the third argument. Once all `distance`'s are consumed, a `min-distance` is derived where the second argument is the minimum distance computed before.

```
select-min(A)
   -o [min => D | B | !edge(A, B), distance(A, B, D) | min-distance(A, D)].
```

3.10 Node Instantiation

When a Meld program starts executing, a static graph model deduced from the static analysis of the source code is instantiated immediately. This is performed by analyzing the source code and looking for node addresses. For a very size-able number of programs this may be enough, but there as large class of programs where the number of nodes is variable and depends on the problem being solved. Meld allows the creation of new nodes through the directive `create` (which mimics the $\exists$ connective in the proof theory of linear logic). This directive is only allowed in the head of rules. The syntax for `create` is the following:

```
body -o create Node1 Node2 ... NodeN. (head1), head2.
```

As we can see, multiple nodes may be instantiated at once. Inside `head1` we may use `Node1` up-to `Node2` to derive new facts in those nodes. `create` does not affect `head2`, therefore the new node variables are not available here.

3.11 Distribution Constraints

We have said before that for every fact, the first argument indicates the node where the fact is located. This helps distribute the facts across the nodes, so that computation happens at the node level, only considering the local facts.

Another important constraint is that the body of rules only allow facts from the same node. Enforcing this constraint allows the language to do computations at the node level without communication with other nodes. However, the head of rules may derive facts in other nodes. We may refer to other nodes using the following mechanisms:

Constant The node address is known so we may use it immediately.

Route facts Predicates that are marked as `route` have one or more arguments of the type `node`. When a fact of a route predicate is used in the body of a rule, we may use any of the node arguments as a destination in the head.

Create nodes As explained in the previous section, when a new node is instantiated, the new address is usable in the head of the rule.

3.12 How Are Rules Applied

Meld is a distributed language. Due to this fact, evaluation is composed of computation at the node level (only considering the current state of the node) and communication between nodes.

Initially, the node axioms are derived. We consider the rules that may be applicable with these axioms. Rules are prioritized by their position in the file, so rules at the beginning of the file are considered before rules at the end of the file.

Given the set of applicable rules, we attempt matching with the rule with the highest priority. If the rule succeeds and linear facts are consumed, we may have to delete some rules from the set of applicable rules given that we no longer have certain facts. On the other hand, new facts may be derived, so we may now have more applicable rules. The process is repeated again, by considering the new set of applicable rules. When no more rules are applicable, computation stops at this node until some another node sends new facts.

For communication purposes, the language only guarantees that facts sent to a given node appear in the same order as they are being sent/derived. However, there is no atomaticity at the rule level, that is, if a node derives two facts in a single rule application, the target node may only see one fact at a time. For computation purposes, the target node may only consider one, both or different facts from different nodes at any given time. This imposes certain constraints in the logic of programs as there are no guarantees about which rules may be applied first.

4 How It All Fits Together: The Quick-Sort Algorithm

In this section, we present a complete Meld program. We will use the well known quick-sort algorithm. The program will receive as argument

a comma-separated list of integers and then will sort them in ascending order.

Conceptually, we start with a single node that contains the initial list. Then, we divide the list in two using the pivot (smaller integers than the pivot in one list and the remaining integers in the other) and then we instantiate two new nodes and send the lists to them. During this recursive process, we create a tree structure where nodes connect by a route linear predicate. When the list is "small enough", the leaf node will sort it locally and then send the results to the parent node. The parent node then merges the two sorted lists and sends the result to the parent, recursively. In the end, we end up with the sorted list at the root node.

The complete program is as follows:

```
type linear route back(node, node).
type linear down(node, list int).
type linear up(node, list int).
type linear sorted(node, node, list int).
type linear buildpivot(node, list int, int, list int, list int).
type linear waitpivot(node, node, node, int).
type linear append(node, list int, list int).
type linear reverse(node, list int, list int, list int).
type linear reverse2(node, list int, list int).

const tosort = str2intlist(@arg1).

down(@0, tosort).

/* some base cases */
down(A, []) -o up(A, []).
down(A, [X]) -o up(A, [X]).

/* sort two item lists */
down(A, [X, Y]), X < Y -o up(A, [X, Y]).
down(A, [X, Y]), X >= Y -o up(A, [Y, X]).

/* list has more than two elements: select pivot and
   build smaller and greater list */
down(A, [X | L]) -o buildpivot(A, L, X, [], []).

/* lists built, pass them down */
```

```
buildpivot(A, [], X, Smaller, Greater)
   -o create B, C. (back(B, A), down(B, Smaller), back(C, A),
         down(C, Greater), waitpivot(A, B, C, X)).

/* building the smaller and greater lists */
buildpivot(A, [Y | L], X, Smaller, Greater),
Y <= X
   -o buildpivot(A, L, X, [Y | Smaller], Greater).
buildpivot(A, [Y | L], X, Smaller, Greater),
Y > X
   -o buildpivot(A, L, X, Smaller, [Y | Greater]).

/* merge sorted lists */
waitpivot(A, NodeSmaller, NodeGreater, Pivot),
sorted(A, NodeSmaller, Smaller),
sorted(A, NodeGreater, Greater)
   -o append(A, Smaller, [Pivot | Greater]).

append(A, L1, L2)
   -o reverse(A, L1, L2, []).

reverse(A, [], L2, L3) -o reverse2(A, L3, L2).
reverse(A, [X | L], L2, L3) -o reverse(A, L, L2, [X | L3]).

/* append is done here */
reverse2(A, [], Result) -o up(A, Result).
reverse2(A, [X | L1], L2) -o reverse2(A, L1, [X | L2]).

/* once we sort the sublist, we pass it up to be merged */
up(A, L), back(A, B) -o sorted(B, A, L).
```

Let's now go through some important parts of this program.

After all the predicates are declared, we define the `tosort` constant:

```
const tosort = str2intlist(@arg1).
```

We use the external function `str2intlist` to convert the first input of the program (a string) to a list of integers. Next, we define the following axiom for the root node using the defined constant:

```
down(@0, tosort).
```

When the fact `down` is derived on a specific node, it signals the node that the list in the second argument must be sorted. After the sorting, the node will derive an `up` fact that signals the node that its list has been sorted and can be sent to the parent node. Depending on the format of the list, we have three possibilities: if the list is empty or has only one element, it remains the same; if the list has two elements, the list is simply sorted; and if the list has more than two elements, we start the recursive sorting process by deriving `buildpivot`.

```
/* some base cases */
down(A, []) -o up(A, []).
down(A, [X]) -o up(A, [X]).

/* sort two item lists */
down(A, [X, Y]), X < Y -o up(A, [X, Y]).
down(A, [X, Y]), X >= Y -o up(A, [Y, X]).

/* list has more than two elements: select pivot and
   build smaller and greater list */
down(A, [X | L]) -o buildpivot(A, L, X, [], []).
```

The predicate `buildpivot` has four arguments (apart from the node): the initial list, the pivot element, the list with numbers smaller than the pivot and the list with numbers greater than the pivot. We then have two rules that build the smaller and greater lists by taking apart the second argument:

```
/* building the smaller and greater lists */
buildpivot(A, [Y | L], X, Smaller, Greater),
Y <= X
   -o buildpivot(A, L, X, [Y | Smaller], Greater).
buildpivot(A, [Y | L], X, Smaller, Greater),
Y > X
   -o buildpivot(A, L, X, Smaller, [Y | Greater]).
```

Finally, when the second argument is empty (the list has been fully traversed), we instantiate two new nodes that will sort each list (smaller and greater) separately:

```
/* lists built, pass them down */
buildpivot(A, [], X, Smaller, Greater)
```

```
   -o create B, C. (back(B, A), down(B, Smaller), back(C, A),
         down(C, Greater), waitpivot(A, B, C, X)).
```

Each `down` fact signals the corresponding node that a sublist must be sorted and `back` links the new nodes to the parent node, so that communication is possible after the sublists are sorted. We also derive `waitpivot` that is used to wait for the results of the sublists.

The following rule matches `up` and `back` so that the results of sorting can be sent to the parent node through the `sorted` fact.

```
/* once we sort the sublist, we pass it up to be merged */
up(A, L), back(A, B) -o sorted(B, A, L).
```

Obviously, there is a rule where we wait for two `sorted` facts using `waitpivot`:

```
/* merge sorted lists */
waitpivot(A, NodeSmaller, NodeGreater, Pivot),
sorted(A, NodeSmaller, Smaller),
sorted(A, NodeGreater, Greater)
   -o append(A, Smaller, [Pivot | Greater]).
```

The fact `append` appends two lists: the "smaller" list with the pivot plus the greater list, generating the final sorted list. Here is the full code:

```
append(A, L1, L2)
   -o reverse(A, L1, L2, []).

reverse(A, [], L2, L3) -o reverse2(A, L3, L2).
reverse(A, [X | L], L2, L3) -o reverse(A, L, L2, [X | L3]).

/* append is done here */
reverse2(A, [], Result) -o up(A, Result).
reverse2(A, [X | L1], L2) -o reverse2(A, L1, [X | L2]).
```

Note that `reverse2` derives `up` once its first argument is consumed. When computation completes, the root node will contain an `up` fact with the initial list sorted in ascending order.