Multidimensional Predicates for Prolog

Günter Khyo
The Nexialist Foundation
(*e-mail:* `guenter.khyo@chello.at`)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

In 2014, Ungar et al. proposed *Korz*, a new computational model for structuring adaptive (object-oriented) systems [UOK14]. Korz combines implicit parameters and multiple dispatch to structure the behavior of objects in a multidimensional space.

We will see how the ideas of Korz can be applied to context-oriented logic programming in Prolog, using a new library called *mdp* (multidimensional predicates). We will explore numerous scenarios such as logging, memoization, object-oriented programming and adaptive GUIs.

In particular, we will see that we can structure and extend Prolog programs with additional concerns in a clear and concise manner. We also demonstrate how Prolog's unique meta-programming capabilities allow for quick experimentation with syntactical and semantic enhancement of the new model.

1 Motivation

From the discoveries of Einstein and Heisenberg we know that absolute descriptions of the universe ultimately lead to contradictions [vF72]. This insight has also forced biologists to rethink theories about living organisms and led to new theories about life which include the observer [MV79].

In the domains of artificial intelligence and knowledge-representation, the notion of context and context-sensitive reasoning has been studied in-depth [McC93, Guh92, Lok04], giving rise to numerous formalisms and programming techniques. Since the advent of mobile computing, there has also been increasing interest in augmenting object-oriented programming languages with contextual models [HO93, vLDN07, AHH^{+}09, KAMT14]. One of the most recent contributions is Korz [UOK14], a model for organizing the behavior of objects (described in methods or slots) in a multidimensional space, where each dimension represents a concern. Every message sent to an object carries a context (a set of concerns) which designates a coordinate, along with the name of the message, that refers to a particular method in the method-space (see Section 5.1 for a small introduction). Unlike other approaches, the conceptual model of Korz is very general, and as we will see, directly applicable to logic-programming, giving birth to multidimensional predicates for Prolog.

Note that the presented work is not a replacement for more expressive models of reasoning within contexts, but a basic, orthogonal augmentation to structure the predicate space of Prolog systems.

2 Introduction

Let us assume we want to implement an expert system for assisting researchers in Prolog. At some point, a researcher might want to inquire the system about the references of a paper. As simple as this problem might seem at first glance, we have to make a number of design choices such as where and how the references are represented (they could be stored as facts, or retrieved from third-party sources such as *dblp*), whether we want to compute all references or just direct references, the fields of expertise the researcher has knowledge of, the citation count of the referenced papers and so on. In essence, the design, or context, space is open. However, there might be fixed scenarios which we would like to model explicitly. For this purpose, we use mdp to organize our contextual space.

In the default case, if we have no contextual information, we decide to ask the Prolog database for direct references for a given paper *Paper*:

```
[] # referenced(Paper, Reference) :-
  lookup_reference(Paper, Reference).
```

A contextualized rule consists of two parts: a list of concerns and an ordinary rule definition separated by a hashtag. The concerns are modeled as key-value pairs, where a key is represented by an atom and the value by an arbitrary term (following the model of Korz, we call them dimensions and coordinates). If we are only interested in particular fields, we can add following rule:

```
[fields: KnownFields] # referenced(Paper, Reference) :-
  lookup_reference(Paper, Reference),

  field(Reference, Field),
  member(Field, KnownFields).
```

Similarly, we might be only interested in papers with a citation count exceeding a given threshold:

```
[citation_count: Threshold] # referenced(Paper, Reference) :-
  lookup_reference(Paper, Reference),

  citation_count(Reference, C),
  C > Threshold.
```

We evaluate a contextualized rule with the *?* operator, e.g.,

```
?- [citation_count: 100] ? reference(P1, P2).
```

The predicate which is most specific to the given context will then be evaluated, i.e., the definitions which have the highest number of matching dimension/value pairs. All stated context dimensions of a rule must be present for the matching result to count. (In addition, we can specify preconditions (clauses) inside the context which then have to hold for the result to count).

For example, when we state the query

```
?- C=[citation_count:50,fields:['computer␣science', 'cybernetics']],
    C ? referenced(P1, P2).
```

our previous two definitions are evaluated, since no one is more specific than the other, i.e., for both definitions there is an equal number of matching dimensions. To resolve this issue, we define a combined rule to decide what happens when all dimensions are present:

```
[citation_count: _, fields: _] # referenced(Paper, Reference) :-
  [-citation_count] ? referenced(Paper, Reference),
  [-fields] ? referenced(Paper, Reference).
```

The context we give to an context-aware evaluation is passed implicitly down the predicate evaluation chain. An expression like `[-citation_count]` means to remove a concern from the implicit context. This allows us to narrow down the context and reuse specific contextualized rules.

We can now use our contextualized-relation to compute all paper references such as:

```
[] # graph(Paper, References) :-
  compute_transitive_closure((Paper, Reference),
           ? referenced(Paper, Reference), References).
```

The context which we pass to *graph/2* is then forwarded to the referenced-relation. The rule is oblivious of the particular choice of dimensions we make for *referenced/2*. We can imagine having different versions of *graph/2*, for example one that computes only direct references. We could also pass a predicate which specifies a traversal on the graph as a dimensional argument which is then evaluated using *call/N* or *apply/N*.

Using mdp, we can also express cross-cutting concerns. For example, we might be interested in logging the evaluation of certain predicates. For this purpose, we can define an anonymous rule as follows:

```
[log_predicates: PL, predicate: P, (...)] :-
  member(P, PL),
  [-log_predicates] ? P,
  write_to_log_file(P).
```

The dimension *predicate* is always present, even if not stated and is bound to the right-hand side of a contextual query, e.g., `referenced(P1, P2)`. The wildcard-expression `(...)` denotes that the rule matches with all non-stated dimension/coordinate pairs. For the definition of the rule, it is important that we remove the *log_predicates* dimension, otherwise the anonymous rule will be evaluated indefinitely.

One might argue that the presented notation is superfluous. This is true in a certain sense, since Prolog already provides all needed language constructs for specifying contexts and multiple definitions of a predicate via unification and backtracking. Listing 1 shows how our little example can be translated into regular Prolog code. However, the translation has following drawbacks:

- The semantics of context selection and manipulation and the semantics of a rule are mixed, making it difficult to experiment with variations of context resolution,
- As such the traditional implementation is also slightly more verbose and less readable,
- The context has to be passed explicitly around.

Also, the presented notation acts as kind of psychological device since it encourages the use of contexts due to its simple and lightweight nature. Finally, I argue that contextual awareness is not merely optional but mandatory for any description, considering the importance it holds for (scientific) descriptions in general. As we will see in the remainder of this paper, there are many useful applications for mdp as shown in Section 3, where we will see how we can model objects, context-sensitive GUIs and memoization. Details on the implementation and guidance on how to extend mdp are found in Section 4.

3 mdp Scenarios

In this section we will see more examples that motivate the use of mdp. We will address following issues: memoization, object-oriented programming and how we can augment a GUI with additional concerns.

3.1 Memoization

Memoization is a common technique used to cache expensive computations. In this section, we will explore how we can orthogonally add the concern of memoization to prime number generation. We will then derive a generic solution to memoize arbitrary predicates.

Suppose we have a naive implementation of a prime number test:

```
[] # is_prime(P) :-
  P > 1,
  UpperTestLimit is floor(sqrt(P)),
  forall(between(2, UpperTestLimit, Divisor), \+ 0 is P mod Divisor).
```

Fortunately we know that, for some reason, there are only a few unknown numbers we are testing so we might get away with memoization. We add following lines:

```
:- dynamic memoized_is_prime/2.

[memoize: _, (...)] # is_prime(P) :-
  memoized_is_prime(P, IsPrime) ->
   call(IsPrime) ;
      ([-memoize] ? is_prime(P) ->
        assertz(memoized_is_prime(P, true)) ;
        assertz(memoized_is_prime(P, false))).
```

This works fine, but if we have to copy and paste the memoization code for every other predicate we would like to enhance with memoization. We remove our specialized implementation and generalize as follows:

```
:- dynamic memoized/2.

[memoize: _, predicate: P, memoized(P, TorF), (...)]  :- call(TorF).

[memoize: _, predicate: P, \+ memoized(P, _), (...)] :-
  forall([-memoize] ? P, assertz(memoized(P, true))),
  (\+ [-memoize] ? P ->
    assertz(memoized(P, false)), fail ;
    memoized(P, _)).
```

This solution is slightly more complex, since it also handles memoization for non-deterministic predicates. Now, for every multidimensional predicate *P* we can state the query:

```
?- [memoize: _] ? P(Args)
```

to memoize. Our solution can be further improved by extending *assert* and *retract* to invalidate memoization data, which is left as an exercise to the reader.

3.2 Object-oriented Programming with mdp

In this section, we will explore how mdp can be used to implement a purely object-oriented system, i.e., no classes. To this end, we built a simple meta-object-protocol *mop* on top of mdp. In the following sections, we explore how we can represent messages and state in mdp and implement a model for subtyping, polymorphism and protecting data.

3.2.1 Objects, Messages and State

In mop, as in Korz, an object is a globally unique identifier. New objects (identifiers) are created with the predicate `new_oid(+NewOID)`. Messages can be sent to objects with the *!* operator (inspired by Erlang), for example, `Display ! render(Object)` which is equivalent to `[rcvr: Display] ? render(Object)`. Thus, a method is just a multidimensional predicate which includes a receiver dimension, containing the id of the object.

The attribute of an object can be inspected and changed with `OID ! read(Name, Value)` and `OID ! write(Name, NewValue)`, respectively. Also, for every object there are predicates to clone the object (which is a shallow copy of its attributes) and to query its type (which, internally, is just an attribute).

The meta-object protocol is defined as follows:

```
:- dynamic data/3. % data(OID, Name, Value)
:- op(0101, xfx, !). % Send a message

[rcvr: OID] # write(Name, Value) :-
```

```
  retractall(data(OID, Name, _)),
  assertz(data(OID, Name, Value)).

[rcvr: OID] # read(Name, Value)  :- data(OID, Name, Value).

[rcvr: OID] # type(T) :- data(OID, type, T).

[rcvr: OID] # clone(OIDClone) :-
  new_oid(OIDClone),
  forall(data(OID, Name, Value), OIDClone ! write(Name, Value)).
```

The message-send operator is implemented using a hook provided by *mdp* (see Section 4):

```
% Translate Receiver ! P(Args) to [rcvr: Receiver] ? P(Args)
hook_mdp_term(
  ImplicitContext,
  Receiver ! Predicate,
  ?(ImplicitContext, [rcvr:Receiver], Predicate)).

% Translate [Context] ? Receiver ! P(Args) to
%            [rcvr: Receiver, Context] ? Receiver ! P(Args)
hook_mdp_term(
  ImplicitContext,
  Receiver ! (?(GivenContext, Predicate)),
  ?(ImplicitContext, [rcvr: Receiver | GivenContext], Predicate)).
```

The last hook-definition allows us to provide a context to a message-send (we will see examples of this in Section 3.2.3 and Section 3.3). (There are other hook-definitions to implement further syntactic variations which can be found in the mdp distribution.)

3.2.2 Subtyping and Polymorphism

Let us consider the canonical example of modeling objects and behaviors for representing geometric shapes. On the type level, we define following subtype-relations:

```
subtype(shape, rectangle).
subtype(rectangle, special_rectangle).
subtype(shape, circle).
```

We define the polymorphic predicate (method) *representation/1* as follows:

```
% OID < shape reads: OID is a subtype of shape
[rcvr: Shape, Shape < shape] # representation(_) :-
  throw('Abstract␣method:␣No␣implementation').

[rcvr: Rectangle, Rectangle < rectangle] # representation(R) :-
  Rectangle ! read(width, W),
  Rectangle ! read(height, H),
  R = rectangle(W, H).

[rcvr: Rectangle, Rectangle < special_rectangle]
 # representation(R) :-
  Rectangle ! read(width, W),
  Rectangle ! read(height, H),
  R = special_rectangle(W, H).
```

```
[rcvr: Circle, Circle < circle] # representation(R) :-
  Circle ! read(radius, Radius),
  R = circle(Radius).
```

Internally, the subtype-relation < is translated to a form mdp can understand, i.e.,

```
[rcvr: OID, OID < rectangle] # representation(R) :- ...

% Translates to

[rcvr: OID,
  [rcvr: OID] ? type(ActualType),
  @(type_affinity(Subtype, ObjectType, D), D)
] # representation(R) :- ...
```

We can specify this translation by defining a rule for a hook supplied by mdp:

```
hook_context_rule_mdp_term(Context, OID < Subtype, Translation) :-
  Translation = (
    [rcvr: OID] ? type(ObjectType),
    @(type_affinity(Subtype, ObjectType, D), D)
  ).
```

where the predicate `type_affinity` measures how close the supplied type constraint fits the actual type of the objects (see Listing 2). The result is then used to give weight to the rule, using the @-annotation, to ensure proper matching.

We can now run the query `AnyShapeType ! representation(R)` and we get the representation corresponding to the type of *AnyShapeType*. Since there are no classes, prototypes have to be defined and cloned to create new instances. For example, the rectangle prototype can be created as follows:

```
?-new_oid(Rectangle),
  Rectangle ! write(type, rectangle),
  Rectangle ! write(width, 100),
  Rectangle ! write(height, 100),

  Rectangle ! representation(R).

R = rectangle(100, 100).
```

There are many improvements we can make to our meta-object protocol. For example, we could check whether an attribute has been declared before writing to it.

3.2.3 Protecting Data

In many class-based object-oriented languages, the programmer may restrict access to certain attributes or methods with visibility modifiers. For example, consider following game:

```
% Overrides default mop read method
[rcvr: OID, OID < guessing_game] # read(secret_number, S) :-
  throw('No␣funny␣stuff').

[rcvr: OID, OID < guessing_game, modifier: private] #
  read(secret_number, S) :-
    number(S),
    ata(OID, secret_number, S).
```

```
[rcvr: OID, OID < guessing_game] # guess(S) :-
  [modifier: private] ? OID ! read(secret_number, S).
```

We could improve our solution by protecting access to *data/3* and providing a cryptographic context which cannot be tempered with. The reader is encouraged to experiment with more secure models.

The key point is that mdp provides all facilities to model a variety of concerns and no special language constructs such as visibility modifiers are needed. In fact, mdp offers even more flexibility, for instance, we can also imagine role-based visibility.

3.3 An Adaptive GUI

In this section we will explore how we can model an adaptive graphical user interface system. (Note that the example is only a sketch and serves for demonstrating the potential of mdp. We will provide full-fledged examples of adaptive GUIs in future works).

The system executes within a dynamic context which can be manipulated by the user or other parts of the system. For the moment, we do not make any assumptions how the context is structured, i.e., which dimensions of user-experience it contains. In this example, we focus on displaying objects only.

In some part of the system, assume we have a predicate *refreshed(Display, Object)* which is evaluated whenever there is a change in the system context or one of its graphical objects:

```
refreshed(Display, Object) :-
  system_context(Context),
  Context ? Object ! representation(R),
  Display ! present(R).
```

Different types of objects have different graphical representations. For the moment, we assume that all objects are represented as colored bitmaps. We use a third-party library for rasterization. There might be following implementations for the predicate *representation*.

```
[rcvr: G, G < text] # representation(R) :-
        G ! text_contents(T),
        G ! text_color(Color),

        text_pixmap(T, Color, R).

[rcvr: G, G < box]  # representation(R) :-
        G ! bounds(Width, Height),
        G ! color(Color),

        box_pixmap(Width, Height, Color, R).

% Specifications for other shape types ...

[rcvr: G, G < object] # representation(R) :-
        default_representation(R).
```

Many modern GUIs frameworks accept different types of graphical representations, for example vector graphics. Assuming we find out that our display object is capable of rendering vector graphics, we do not have to change any of the existing code to support this. We just add the dimension *render_type* to our system (and provide means of render-type-selection for the user) and add a corresponding predicate definition:

```
[rcvr: G, G < box, render_type: svg] # representation(R) :-
        G ! bounds(Width, Height),
        G ! color(Color),
        R = svg(shape=box, color=Color).

[rcvr: G, G < box, render_type: pixmap] # representation(R) :-
        % Fallback to original implementation
        [-render_type] ? G ! representation(R).
```

Now suppose that there is a light-sensor hooked to our system and we would like to take advantage of this by adapting the colors of the objects to the ambient light. For example, when the room is dark, we want to display our objects in a midnight blue theme. Now, we could add following lines:

```
[rcvr: G, G < text, ambient_light: dark] # representation(R) :-
        G ! colored(gray),
        [-ambient_light] ? G ! representation(R).

[rcvr: G, G < box, ambient_light: dark] # representation(R) :-
        G ! colored(midnight_blue),
        [-ambient_light] ? G ! representation(R).
```

Unfortunately, this will introduce non-determinism if the context contains both an *ambient_light* and a *render_type* dimension. The reason is that, given a graphical object, the light-sensitive and the render-type sensitive implementations of *representation* match the context as none is more specific than the other. *mdp* will evaluate both variants, in the order of their definitions.

```
?- box_prototype(T),
   [ambient_light: dark, render_type: svg] ? T ! representation(R).

  R = svg(shape=box, color=midnight_blue) ;
  R = svg(shape=box, color=original_color).
```

We can fix this in two ways: We can either add the `render_type` dimension to our definitions or we can assign a higher score to the context:

```
[rcvr: G, G < box, ambient_light: dark,
 render_type:_] # representation(R) :-

        G ! set_color(midnight_blue),
        [-ambient_light] ? G ! representation(R).

% Alternative implementation

dimension_weight(ambient_light, 2).

[rcvr: G, G < box,
 ambient_light: dark,
 dimension_weight(ambient_light, W)@W] # representation(R) :-
        G ! set_color(midnight_blue),
        [-ambient_light] ? G ! representation(R).
```

However, both solutions have the problem that all variants of representation have to be examined to safely add new variants. Nonetheless, the latter solution of assigning weights to dimension, gives us more flexibility as it allows us to organize the dimensional weights as a separate concern. Nonetheless, the programmer might forget that there is a weight assignment for a specific dimension and manually adding a weight annotation to each dimension is too tedious. However, it is straightforward extend mdp with automatic annotations, derived from a designated weight assignment predicate.

The bigger issue is how the programmer can be assisted in structuring a multidimensional predicate space. Also, Ungar et al. stress the importance of having equal dimensions ("'no dimension holds sway over another"'). Future research and experimentation with larger systems is needed to assess this problem.

4 Implementation

The implementation consists of two parts: an extensible transformer which translates mdp specifications into ordinary Prolog rules and a dispatcher which is responsible for evaluating multidimensional queries by selecting and evaluating the most specific predicates w.r.t. a given a context.

Both the transformer and the dispatcher are implemented in standard Prolog. The transformer implements the predicate *term_expansion/3* which is offered by Prolog systems for rewriting consulted programs.

4.1 Transforming mdp-Code

For every mdp rule, the transformer generates two parts, an implementation for the predicate and a signature which contains various meta data used by the dispatcher to determine which implementation to select.

For example, consider following mdp rule:

```
[debug: P, weight(debug, D)@D] # edge(A, B) :-
   [-debug] ? edge(A, B),
   call(P, (A, B)).
```

The transformer will generate following code:

```
mdp_signature(
  edge/2,
  mdp_implementation(edge/815),
  % where 815 is a generated identifier to distinguish
  % mdp predicates that have the same functor and arity

  context_spec(
    % Context variable
    Ctx,

    % Defined dimensions
    [debug],

    % Translated context specification (used by the dispatcher)
    [ctx_member(Ctx, debug, P), weight(debug, D)],

    % Matching-score variables (summed up by the dispatcher),
    [D]
  )
).

mdp_implementation(edge/815, Ctx, A, B) :-
  ctx_member(Ctx, debug, P),
  ? (Ctx, [-debug], edge(A, B), % invoke dispatcher
  call(P, (A, B)).
```

The basic transition rules for the transformer and the semantics of mdp are shown in Table 1.

The transformer is applied to every mdp definition and executes within an environment Γ. Γ comprises four kinds of information: $\Gamma_{context}$, $\Gamma_{dimensions}$, Γ_{scores} and Γ_{rules}. $\Gamma_{context}$ denotes a fresh variable which holds the context of an mdp rule. Initially, it contains the dimensions *predicate*. $\Gamma_{dimensions}$ and Γ_{scores} are initially empty lists that are extended by the transition rules to contain the required dimension names and the weights of an mdp rule. The weights are either variables or integer constants. Finally, Γ_{rules} contains the translated code of a context-specification. For readability reasons its derivation is omitted.

The rules can be extended to accommodate for syntactical and semantical enhancements such as we have seen for object-oriented programming (see Section 3.2). For this purpose, the transformer provides two hooks that allow the extension of the basic rules.

The first hook *hook_context_rule_mdp_term(+Context, +Rule, -Term)* is applied to every term in a context specification and can be extended by the programmer to add transformation rules. *Term* can be either ordinary Prolog code or an mdp query.

The second hook *hook_mdp_term(+Context, +Term1, -Term2)* is applied after the first hook and transforms the context specification terms and the bodies of predicates into ordinary prolog terms.

4.2 Predicate Selection and Evaluation

When given a multidimensional query, the dispatcher will manipulate the implicit context (such as remove or change dimensions) and select the most specific predicates (MSPs).

The MSPs are determined by following steps:

1. Collect the list of all defined mdp rules that match the functor and arity of a query.
2. Add all anonymous rules to the list.
3. Remove all rules which have failed preconditions.
4. Remove all rules which require more dimensions than are present in the implicit context.
5. Determine the score of every rule by counting one for each matched dimension and adding up all weights. In addition, if there is a wildcard expression, count one for every dimension given in an implicit context, except the ones which are explicitly stated.
6. Select all predicates with the highest score.

Listing 3 shows how the dispatcher works in detail.

5 Related Work

mdp is a derivative work of Korz. It introduces some alterations to the original model (see Section 5.1) such as selector-based matching, weights, wildcards and using Prolog, can exploit pattern matching on dimension coordinates (which is not possible in the Korz model).

5.1 Korz: Background and Introduction

In their critique on pure objects, William Harrison and Harold Ossher note that the classical model of object-orientation which demands one objective definition of an object is "‘inadequate to deal with the construction of large and growing suites of applications manipulating the objects"’, since "‘designers are forced to either forego the advantages of object-oriented style or to anticipate all future applications"’ [HO93].

According to Harrison and Ossher, properties and behavior are not intrinsic to an object, but *extrinsic*, as they emerge from particular points of view. They propose a detailed subject-oriented programming model for class-based OO systems.

Recently, David Ungar et al. together with Ossher developed a system for context-oriented programming (subsuming subjectivity) on top of *Self* [US87], called *Korz* [UOK14]. In Self, an object is collection of slots, where each slot either holds a method, comprising sequences of messages sent to other objects or the object containing the slot, or a reference to another object. Every slot is reachable via a path from an object, expressed by a sequence of selectors each matching a slot name. For example by sending the message `position x` (with selectors *position* and *x*) to a geometric object.

To accommodate for contextual requirements, Ungar et al. abandoned the notion of objects as rigid organizations of slots and propose a more flexible, multidimensional slot space model. In contrast to Self, a slot does not belong to a particular object. Instead, its location is defined by its dimensions and coordinates, formulated within *slot guards*. Every dimension represents a particular concern in the system and coordinates are values that refer to objects (which are represented as globally unique identifiers).

In the zero-dimensional case, when the slot guard of a method slot is empty, we can think of it as a "‘global"’ procedure. For example, we can envision having a procedure (slot) for drawing a pixel, with an empty slot guard (`{}`).

```
method {} drawPixel(x, y, color, screen) {
        // draw the pixel in the color on screen
}

drawPixel(...);
```

In object-oriented designs, drawPixel is usually bound to an object. In Korz, this can be expressed by adding a receiver (`rcvr`) dimension:

```
def {} screen = newCoord;

// Receiver has to be a subtype (<) of screen for this slot to match
method {rcvr < screen} drawPixel(x, y, color) {
        drawPixel(x, y, color, screen)
}

// Assign screen coordinate to the receiver dimension
{rcvr: screen}.drawPixel(...) ;
```

To see how our screen looks like to blind people, we can define a specialized drawPixel method which converts its supplied color value into a grayscale value:

```
method { rcvr < screenParent, isColorblind < true }
drawPixel(x, y, c) {
        // Remove colorblind dimension
        {-isColorblind}.drawPixel(x, y, c.mapToGrayScale)

        // The binding for receiver is carried along implicitly,
        // i.e., the above call is equivalent to
        // {rcvr: screen, -isColorblind}.drawPixel(...)
}

{rcvr: screen, isColorblind: true}.drawPixel(...);
```

Thus, using slot guards, multiple version of a slot can coexist. The slot which is most specific with regards to the context (the supplied coordinate-value pairs), will be selected.

5.2 Differences to Korz

While mdp rests on a Prolog, a single-dimensional language, its unique properties of uniformity, minimality and homoiconicity provide a flexible foundation for embedding multidimensionality and experimenting with new features. This allowed me to quickly implement one of Ungar et al.'s suggested improvements to Korz, selector-based matching (anonymous rules) and alterations of my own accord such as wildcards.

Also, unlike Korz, multiple methods (predicates) can be executed if they have the same specificity. In Korz this is strictly forbidden. I made this design choice because non-determism is an important and desired feature of Prolog programming and it naturally extends to multidimensional predicates.

Also, we can exploit Prolog's unification capabilities to perform pattern matching on dimension coordinates. For example, we could match on a profiling dimension as follows:

```
[profile: (ExecutionTime, P), predicate: P, (...)] :-
  get_time(T0),
  [-profile] ? P,
  get_time(T1).

-? [profile: P] ? predicate(Arguments), writeln(P).
```

Another notable difference is mdp's reliance on weights which allow us to influence predicate selection. The initial design did not include weights, but when introducing subtyping, it became apparent that there must be a kind of weight dimension.

6 Conclusion and Future Work

We have seen that we can use mdp for many purposes. We can use multidimensional predicates to do object-oriented programming and to extend a program with additional concerns. Using techniques such as memoization, we obtain the benefit of having readable and efficient specifications. Yet we have barely scratched the surface and we will present more scenarios in future works.

In particular, there are following issues that need attention:

- **Efficiency of Implementation**
  A major drawback of mdp is its efficiency. The dispatching mechanism for multidimensional predicates is very inefficient and has been programmed with quick adaptability and readability in mind. Its performance is unlikely to be satisfactory in production systems. For satisfactory performance, a Prolog VM could be extended with special instructions for multidimensional dispatching.
  However, it is not clear at this stage whether the mechanism is sufficiently expressive for large-scale Prolog systems. To this end, more scenarios have to be investigated to reveal possible shortcomings.
- **Debugging and Comprehensibility**
  Currently, there is no support for debugging mdp programs. When debugging an mdp program, the programmer has to trace the call-chain of the mdp implementation to get information such as the context and predicate selection. A usable debugger has to hide these implementation details from the user and present the desired information in an easily accessible way.
  Also, the programmer has to be very careful in organizing multidimensional predicates. When programming in the large-scale, a development environment that helps a programmer organizing dimensions and point out at possible mistakes is mandatory.
  The design of such a development environment with a suitable debugger will be an interesting challenge.
- **Extending the mop**
  The meta-object protocol we presented is far from complete. For example, we can enrich mop with syntactical sugar for defining OOP-style properties and context-oriented visibility modifiers. Also, the syntax for matching subtypes is a bit verbose, i.e., `[rcvr: OID, OID < Subtype]` can be shortened to `[OID < Subtype]` (like in Korz). Modifying mop accordingly is straightforward.
- **Restructuring Prolog Libraries**
  mdp gives us many new ways to structure Prolog code and much incentive to restructure existing code bases. For example, the `ordsets`-library provides special predicates for manipulating and querying data types in ordered sets such as *ord_intersection/3* or *ord_memberchk/2*. With mdp we can give the programmer a more consistent experience and have overloaded predicate names such as *member* and, depending on the context, dynamically choose the predicates to deal with special data types.
  Another interesting experiment would be to rewrite XPCE [WA02], a GUI framework for SWI-Prolog, using the techniques we have presented in this paper.

I welcome suggestions, bug reports and improvements to the implementation which can be obtained at: `https://bitbucket.org/nexialist/mdp`.

References

Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming*, COP '09, pages 6:1–6:6, New York, NY, USA, 2009. ACM.

Ramanathan Guha. *Contexts: A Formalization and Some Applications*. PhD thesis, Stanford, CA, USA, 1992. UMI Order No. GAX92-17827.

William Harrison and Harold Ossher. Subject-oriented programming: A critique of pure objects. *SIGPLAN Not.*, 28(10):411–428, October 1993.

Tetsuo Kamina, Tomoyuki Aotani, Hidehiko Masuhara, and Tetsuo Tamai. Context-oriented software engineering: A modularity vision. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 85–98, New York, NY, USA, 2014. ACM.

Seng W. Loke. Logic programming for context-aware pervasive computing: Language support, characterizing situations, and integration with the web. In *Proceedings of the 2004 IEEE/WIC/ACM International Conference on Web Intelligence*, WI '04, pages 44–50, Washington, DC, USA, 2004. IEEE Computer Society.

John McCarthy. Notes on formalizing context. In *Proceedings of the 13th International Joint Conference on Artifical Intelligence - Volume 1*, IJCAI'93, pages 555–560, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

Humberto R. Maturana and Francisco J. Varela. *Autopoiesis and Cognition: The Realization of the Living*. Boston Studies in the Philosophy and History of Science, 1979.

David Ungar, Harold Ossher, and Doug Kimelman. Korz: Simple, symmetric, subjective, context-oriented programming. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 113–131, New York, NY, USA, 2014. ACM.

David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM.

Heinz von Foerster. Notes on an epistemology for living things. *BCL Report, No. 9.3*, 1972.

Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: Beyond layers. In *Proceedings of the 2007 International Conference on Dynamic Languages: In Conjunction with the 15th International Smalltalk Joint Conference 2007*, ICDL '07, pages 143–156, New York, NY, USA, 2007. ACM.

Jan Wielemaker and Anjo Anjewierden. Programming in XPCE/Prolog, 2002. Available online: http://info.ee.pw.edu.pl/Prolog/Download/userguide.pdf, Accessed: 07.03.2016.

7 Appendix

```
referenced(Context, P1, P2) :-
  get_dict(citation_count, Context, Threshold),
  \+ get_dict(fields, Context, _),

  lookup_reference(P1, P2),
  citation_count(P2, C),
  C > Threshold.

referenced(Context, P1, P2) :-
  get_dict(fields, Context, KnownFields),
  \+ get_dict(citation_count, Context, _),

  lookup_reference(P1, P2),
  field(P2, Field),
  member(Field, KnownFields).

referenced(Context, P1, P2) :-
  get_dict(fields, Context, _),
  get_dict(citation_count, Context, _),

  del_dict(fields, Context, _, C1),
  referenced(C1, P1, P2),

  del_dict(citation_count, Context, _,  C2),
  referenced(C2, P1, P2).
```

Listing 1: Context-oriented definition of referenced/2 using SWI-Prolog dictionaries

```
type_affinity(T, S, N) :-
  max_type_distance(D),
  type_distance(T, S, DistanceTS),
  N is D - DistanceTS + 1.

type_distance(T, T, 1).

type_distance(T, S, N) :-
  T \= S,
  subtype(Parent, S),
  type_distance(T, Parent, N1),
  N is N1 + 1.

max_type_distance(D) :-
  findall(D, (subtype(T, _),
              subtype(_, S),
              type_distance(T, S, D)), Distances),
  max_list(Distances, D).
```

Listing 2: Measuring type distance

```
% GivenContext refers to the explicit context
% given in a multidimensional query
dispatch(ImplicitContext, GivenContext, Predicate) :-
  % Update context information (remove or update dimensions)
  updated_context(
   ImplicitContext,GivenContext, Predicate, UpdatedContext),

  Predicate =.. [_|Arguments],
  most_specific_predicate(UpdatedContext, Predicate, MSP),
  evaluate(MSP, UpdatedContext, [UpdatedContext|Arguments]).

most_specific_predicate(Ctx, Predicate, MSP) :-
  functor(Predicate, Name, Arity),

  findall(ScoreKey-MSP, ((
    mdp_signature(Name/Arity, MSP, S) ;
    mdp_signature('$anonymous_rule'/_, MSP, S)
   ),

   predicate_score(Ctx, S, Score),
   ScoreKey is -Score), MSPByScore),

   keysort(MSPByScore, [-(Score, _)|_]),
  member(Score-MSP, MSPByScore).

predicate_score(Context, Signature, Score) :-
  Signature =
    context_spec(Context, RequiredDimensions, ContextRules,
                                              Weights),
  ctx_keys(Context, GivenDimensions),

  % All required dimensions have to be present
  intersection(RequiredDimensions, GivenDimensions ,
                                RequiredDimensions),

  % Count the number of matching dimensions, taking
  % wildcards into consideration
  num_matching_dimensions(RequiredDimensions, GivenDimensions
                          NumMatchingDimensions),

  call(ContextRules), % Evaluate preconditions
  sum_list(Weights, WeightScore),
  Score is WeightScore + NumMatchingDimensions.

evaluate(MSP, _, Arguments) :-
  MSP \= mdp_implementation('$_anonymous_rule'/_),
  apply(MSP, Arguments).

evaluate(MSP, Ctx, _) :-
  MSP=mdp_implementation('$_anonymous_rule'/_),
  apply(MSP, [Ctx]).
```

Listing 3: Dispatcher Implementation

(Phase 1) The following rules are applied to every term in an mdp rule

$$\text{multidimensional query}\;\frac{\Gamma \vdash \mathit{Context}?P(\mathit{Args})}{\Gamma \vdash \mathit{dispatch}(\Gamma_{\mathit{context}}, \mathit{Context}, P(\mathit{Args}))}$$

$$\text{term extension}\;\frac{\Gamma \vdash T, \mathit{hook\_mdp\_term}(\Gamma_{\mathit{context}}, T, T')}{\Gamma \vdash T'}$$

(Phase 2) Translation of mdp rules to single-dimensional rules

$$\text{dimension}\;\frac{\Gamma \vdash [\mathit{Dim} : \mathit{Coord} \,|\, R] \# P(\mathit{Args}) :\!- \mathit{Body}}{\Gamma, \Gamma_{\mathit{dimensions}} \cup \{\mathit{Dim}\} \vdash R \# P(\mathit{Args}) :\!- \mathit{ctx\_member}(\Gamma_{\mathit{context}}, \mathit{Dim}, \mathit{Coord}), \mathit{Body}}$$

$$\text{wildcard}\;\frac{\Gamma \vdash [(\ldots) \,|\, R] \# P(\mathit{Args}) :\!- \mathit{Body}}{\Gamma, \Gamma_{\mathit{dimensions}} \cup \{\$\_\mathit{wildcard}\} \vdash R \# P(\mathit{Args}) :\!- \mathit{Body}}$$

$$\text{precondition}\;\frac{\Gamma \vdash [\mathit{Pre}(\mathit{Args}_p) \,|\, R] \# P(\mathit{Args}) :\!- \mathit{Body}}{\Gamma \vdash R \# P(\mathit{Args}) :\!- \mathit{Pre}(\mathit{Args}_p), \mathit{Body}}$$

$$\text{weight}\;\frac{\Gamma \vdash [\_@W \,|\, R] \# P(\mathit{Args}) :\!- \mathit{Body}}{\Gamma, \Gamma_{\mathit{scores}} \cup \{W\} \vdash R \# P(\mathit{Args}) :\!- \mathit{Body}}$$

$$\text{spec extension}\;\frac{\begin{array}{c}\Gamma \vdash [\mathit{SpecPart} \,|\, R] \# P(\mathit{Args}) :\!- \mathit{Body} \\ \mathit{hook\_context\_rule\_mdp\_term}(\Gamma_{\mathit{context}}, \mathit{SpecPart}, \mathit{SpecPart}')\end{array}}{\Gamma \vdash R \# P(\mathit{Args}) :\!- \mathit{SpecPart}', \mathit{Body}}$$

$$\text{rule generation}\;\frac{\Gamma \vdash [\,] \# P(\mathit{Args}) :\!- \mathit{Body}}{\begin{array}{c}\mathit{mdp\_implementation}(P/\mathit{NewID}, \mathit{Args}, \mathit{Body}) \\ \mathit{mdp\_signature}(P/\mathit{Arity}, \mathit{mdp\_implementation}(P/\mathit{NewID}), \\ \mathit{context\_spec}(\Gamma_{\mathit{context}}, \Gamma_{\mathit{dimensions}}, \Gamma_{\mathit{rules}}, \Gamma_{\mathit{scores}}))\end{array}}$$

$$\text{fact extension}\;\frac{\Gamma \vdash \mathit{ContextSpec} \# P(\mathit{Args})}{\Gamma \vdash \mathit{ContextSpec} \# P(\mathit{Args}) :\!- \mathbf{true}}$$

$$\text{anonymous rule}\;\frac{\Gamma \vdash \mathit{ContextSpec} :\!- \mathit{Body}}{\Gamma \vdash \mathit{ContextSpec} \# \text{'\$anonymous\_rule'}(\mathit{Args}) :\!- \mathbf{true}}$$

Table 1: Overview of mdp transformation rules. Note that transition rule "'spec extension"' might require re-application of Phase 1.