A Simple, Symmetric, Subjective Foundation for Object-, Aspect- and Context-Oriented Programming

Harold Ossher David Ungar Doug Kimelman
IBM Research
Yorktown Heights, NY, USA
{ossher,davidungar,dnk}@us.ibm.com

Abstract

Korz is a new computational model that combines implicit arguments (constituting context) with multiple dispatch, in a slot-based model. Slots from a multidimensional space are drawn together into "objects" subjectively, in different ways in different contexts. The combination of these few concepts powerfully supports evolution and contextual variation along multiple dimensions. Korz holds promise as a foundation for future languages that deeply integrate object-oriented, aspect-oriented and context-oriented capabilities. Nonetheless, a number of issues remain to be resolved.

1. Introduction

Korz [Unga14] introduces a new computational model that combines implicit arguments, which serve as context, and multiple dispatch, designed to address multidimensional contextual variation. In addition, rather than a fixed organization of method and data slots into objects, a Korz program fundamentally consists of a multidimensional space of slots, and slots can be drawn together into "objects" subjectively, in different ways in different contexts. There is no dominant decomposition, and no dimension holds sway over any other. At each computation step, multiple dispatch involving both implicit arguments (context) and explicit arguments is used to find the appropriate slot to evaluate.

Though these base concepts individually have been known and used for many years, this particular combination is novel and seems especially powerful: early experience shows that it enables the writing of software that supports contextual variation along multiple dimensions, and graceful evolution of that software to support new, unexpected dimensions of variability, without the need for additional mechanism such as layers [Hirs08] or aspects [Kicz97].

The goal of this paper is to provoke consideration and discussion of the position that the Korz model is a good foundation for future work on languages that deeply integrate object-, aspect-[Elra01] and context-oriented [Hirs08] capabilities. The next section briefly outlines the Korz model, and is followed by discussion of some of the most challenging issues that remain outstanding. More details of Korz, and analysis of related work, are in [Unga14].

2. Overview of Korz Concepts

This section introduces basic Korz concepts and terminology using a simple point drawing example shown in Figures 1 through 3.

In place of an object that constitutes identity as well as a set of slots, Korz has a *coordinate* that is solely a value that constitutes an identity; and instead of being contained by a single object, a slot pertains to a number of coordinates, as indicated by part of its *slot guard* (the slot guard also includes a selector and a list of explicit positional parameters).

In the example shown in Figures 1 through 3, examples of coordinates include: that referred to by the literal 20.0, the contents of the constant slots true and pointParent, and the contents of the variable slot point1. In Figure 2, an example of a slot guard is: { rcvr ≤ screenParent, grayScale ≤ true } drawPixel(x, y, c) , which indicates that the method being defined is contained in a slot with selector drawPixel, has explicit positional parameters x, y, and c, and pertains to coordinate screenParent (which we say is "in the role of" rcvr, or "along the dimension" rcvr) and coordinate true (along the grayScale dimension); that is, the slot is constrained to only be accessible (eligible for selection by the dispatch algorithm) from contexts in which the coordinate in the rcvr dimension is screenParent, or a descendant thereof, and the coordinate in the grayScale dimension is true (any additional dimensions of the context are irrelevant to the accessibility of this slot).

In Korz, a message send occurs in a *context* consisting of a number of coordinates, each in a particular role (or "along a dimension"). The context, selector, and explicit positional arguments of the message send determine the slot to be evaluated. In Figure 1, { rcvr: point1 }.display is an example of a message send (which has no explicit arguments). The context for the message send will include the coordinate point1 in the rcvr dimension, and depending on the chain of sends leading up to this send, the context might also implicitly include a coordinate in the device dimension: stdscreen for example. In this case, syntactic sugar can reduce the code for this message send to point1.display. A message send can add bindings to the context that is being implicitly passed along the call chain, as in { rcvr: f1, device: s, location: australia }.display in Figure 3, which adds three bindings to the context before finding and evaluating a display slot; and a message send can remove bindings from the context, as in
{ -location }.drawPixel(x, -y, c) in Figure 3, which removes any binding of the location dimension to a coordinate.

A slot whose guard has no constraints on context, as in {} pointParent in Figure 1, is globally accessible; i.e., the pointParent slot is not constrained at all with respect to the contexts from which it is accessible, so it is accessible to

```
def {} pointParent = newCoord;
def {} point = newCoord extending pointParent;

var {rcvr ≤ point} x;
var {rcvr ≤ point} y;
var {rcvr ≤ point} color;

method {} makeAPoint(x, y, c) {
   var x, y, c, p;
   p = point.copy;
   p.x = x; p.y = y; p.color = c;
   return p;
}

method {
   rcvr ≤ pointParent,
   device //dimension required but can be anything
   }
   display {
      device.drawPixel(x, y, color)
};


def {} screenParent = newCoord;
def {} screen = newCoord extending screenParent;

method {rcvr ≤ screenParent} drawPixel(x, y, color) {
      // draw the pixel in the color
}


method {} main() {
   var point1 = makeAPoint( 20.0, 30.0, blue );
   point1.display
   // Equivalent to: { rcvr: point1 }.display
   // Context from environment might yield:
   //    { rcvr: point1, device: stdscreen }.display
}
main();
```

Figure 1. Drawing points on a screen.

messages sent from any context. Further, note that if a dimension is mentioned in a guard, but is not constrained to any particular coordinate, as for the device dimension in the guard for the display method in Figure 1, then that dimension must be present in the context for that slot to be accessible, but any coordinate binding is acceptable. As well, in the body of a method, the coordinate to which a dimension is bound may be referred to using the dimension name, as in device.drawPixel(x, y, color) in Figure 1.

The code in the figures is structured using a pattern pioneered in Self [US87], in which prototype objects define data slots, and new objects are created by copying prototypes, which gives them their own data slots. Method slots are defined in the parent of the prototype, which also becomes the parent of the new objects when the prototype is copied. The methods are thus inherited by all the copies.

```
def {} figureParent = newCoord
def {} figure = newCoord extending figureParent;

var {rcvr ≤ figure} point1;
var {rcvr ≤ figure} point2;
var {rcvr ≤ figure} point3;

method {rcvr ≤ figureParent} display {
   point1.display; point2.display; point3.display
}
method { rcvr ≤ screenParent, grayScale ≤ true }
drawPixel(x, y, c) {
   {grayScale: false}
      .drawPixel(x, y, c.mapToGrayScale)
}

   ...
   { rcvr: f1, device: s, grayScale: true }.display
   ...
```

Figure 2. Drawing figures in grayscale.

```
def {} locationParent = newCoord;
def {} location = newCoord extending locationParent;
def {} southernHemi = newCoord extending location;
def {} australia = newCoord extending southernHemi;
def {} antarctica = newCoord extending southernHemi;

method { rcvr ≤ screenParent, location ≤ southernHemi
}
drawPixel(x, y, c) {
   { -location }.drawPixel(x, -y, c)
}

   ...
   { rcvr: f1, device: s, location: australia }.display
   ...
```

Figure 3. Drawing figures inverted.

Please see [Unga14] for a more detailed walk through of the example code, including the steps by which it evolved, as well as a Korz language definition.

3. Discussion: Power and Challenges

Multi-dimensional, symmetric models are powerful, but pose some interesting challenges. Many are due to the principle that dimensions be treated equally: none should be dominant or special, and they should be unordered. The following subsections suggest how the power of the fundamental approach might be applied to dealing with some of the most vexing of these challenges. We feel that these possibilities constitute intriguing directions for future research.

Dimensions for Interpreter Control

A number of the challenges highlighted below can be addressed by providing some direction to the interpreter. One

approach we have begun to explore for providing such direction is *interpreter-control dimensions* that affect the functioning of the interpreter. An example would be a dimension that controls the handling of failure: depending on the value in that dimension, a lookup failure could result in different actions, such as bringing up a debugger, logging the error and terminating the program, or executing context-specific code provided by the developer. Thus, the power of being able to introduce new dimensions for new purposes is used below to facilitate graceful language evolution, not only program evolution.

The 'rcvr' dimension

The dimension rcvr in the example is analogous to the 'receiver' or 'this' object of object-oriented languages, and is thus familiar to programmers. It is at odds with two Korz principles, however: that a single receiver is replaced by a multidimensional context, and that all dimensions are treated equally.

It might seem better in the example to have chosen some other dimension name, perhaps graphic to indicate that it deals with a graphic object. Then the x slot, for example, would have been defined as var {graphic ≤ point} x. There are two problems, however: The first has to do with methods like copy, built-in or library methods that apply broadly. Such a method must use some dimension for the implicit parameter it operates on (such a parameter would be the receiver in object-oriented languages), and since the possibilities for such a parameter are so broad and generic, a domain-specific dimension name like graphic would not suit. We could use a dimension name such as object, entity, thing or the like, but wanted to avoid confusion between coordinates and objects, and also avoid the implication that objects occur in only one particular dimension. Another possibility might be id or identity, but all coordinates in all dimensions are identities. So we chose rcvr, to be suggestive of the object-oriented receiver.

One possible solution would be to define methods like copy as global methods that take an explicit parameter: Instead of method { rcvr } copy() { ... } define method {} copy(x) { ... } . This approach breaks down, however, for methods associated with abstractions like collections, where the use of explicit parameters becomes clumsy and counter to the expectations of object-oriented programmers. In such cases we could possibly use other appropriate dimensions, like collection, rather than rcvr.

That leads to the second problem: the need to switch between dimensions, and its impact on syntactic sugaring. Suppose we had used the graphic dimension as suggested earlier. The makeAPoint method would now have to be written:

```
method {} makeAPoint(x, y, c) {
  var x, y, c, p;
  p = point.copy;
  {graphic: p}.x = x; {graphic: p}.y = y;
  {graphic: p }.color = c;
  return p;
}
```

This is clumsy, and it gets much worse in the case of cascaded expressions. The syntactic sugaring allows one to write p.x = x and so on instead, which is much clearer, and does exactly what an object-oriented programmer would expect. This sugaring, of course, relies on its being clear what dimension is involved. In our current implementation, that dimension is always assumed to be rcvr, and this is the one respect in which rcvr is treated specially. We have begun considering a construct that would allow the programmer to specify the dimension to use, which would allow makeAPoint to be written something like:

```
with implied dimension = graphic {
  method {} makeAPoint(x, y, c) {
    var x, y, c, p;
    p = {rcvr: point}.copy;
    p.x = x; p.y = y; p.color = c;
    return p;
  }
}
```

Now, unfortunately, the copy message can no longer be sugared, because it uses a different dimension (whether rcvr or something else), but, on balance, this might be a better way to write this particular method. We are also interested in IDE support that allows the same code to be viewed in different ways, including with different choices of implied dimension and consequent sugaring.

Lookup Specificity

As discussed in [Unga14], a situation sometimes arises where a message matches two slots, one of which has more dimensions in its slot guard but less-specific coordinates in some of the dimensions that the guards have in common. We opted to define the slot guard with more dimensions as being more specific, irrespective of the coordinates in the common dimensions. This decision supports the important evolution scenario of adding new dimensions to an existing system. However, it can be problematic in the case of evolution involving specialization. An interpreter-control dimension could allow for context-sensitive determination of specificity, but might reduce code comprehensibility.

Super

In any specificity-based dispatch approach, it is important for a method to be able to trigger invocation of the next-most-specific method (e.g. via super or call-next-method). This is a known difficult problem in multiple-dispatch languages, because there is often not a natural, unique next-most-specific method. Rules have to be employed, such as linearization in CLOS [Gabr91] and all-parents-equal in Self, or the programmer can be explicit, as in C++ and Self.

The same issues arise in Korz. They are particularly challenging due to our desire to preserve symmetry and hence avoid any sort of built in linearization or dimension ordering. An interpreter-control dimension could be applied, allowing the context to specify the order in which dimensions should be relaxed, to provide context-specific code to be executed, or to effect some other disambiguation scheme. As in the case of lookup specificity, this degree of dynamicity could reduce code comprehensibility.

Modularity and AOSD

Dimensions provide a flexible and powerful modularization mechanism, that can be used for program organization and presentation and also has presence at runtime, which is important in dynamic languages. A module can be represented by a specific dimension, or a coordinate within a specific dimension. In the first case, the slots to be encapsulated must mention that dimension in their guards. In the second case, the guards must constrain the dimension to the appropriate coordinate. If either of these approaches is followed, slots will be modularized and be inaccessible from other modules unless the context is explicitly set up to have the appropriate dimension bindings.

This means that the core mechanism of Korz addresses many of the evolution scenarios that were the driving force of the aspect-oriented software development (AOSD) community [Elra01], without the need for additional concepts such as aspects, subjects or layers. Obliviousness is supported because slots can be added as described above without pre-existing slots needing any knowledge of them. There are, however, two limitations with respect to AOSD

The first limitation has to do with *method combination*. When adding functionality to a system, it is often necessary to add behavior to an existing method (in addition to adding or replacing methods). This addition is done by such mechanisms as advice weaving in AOP [Kicz97] or composition in SOP [HO93] and MDSOC [Tarr99], which effectively allow multiple methods to be executed in response to a single message, suitably orchestrated. Korz, on the other hand, always selects a single, most-specific method to execute, and fails if there isn't one. An interpreter-control dimension could support other, context-specific options, such as Ensembles [UA10], which run every matching slot for this message, or execution of some context-specific "meta-code" that coordinates execution of the matching slots. In addition, and possibly in conjunction, the "super" mechanism described above can be used to achieve the effect of around advice.

The second limitation of Korz relative to AOSD is that its support for *pointcuts* (specifying in one place code that is to be executed at multiple sites, or *join points*) is limited to what can be done with inheritance. Multidimensional inheritance with dynamic parents is powerful. If an extension provides additional methods for some existing situation guarded by coordinates in one or more new dimensions, then, provided those dimensions are bound to suitable coordinates in the context, the new methods will match the original message, and hence be candidates for combination as determined by the interpreter-control dimension above.

However, the case that is not covered by this is that of selector-based matching: for example, adding behavior to all methods with a particular selector, irrespective of the other elements of their guards. This case can be handled by the kind of extension to Korz that is discussed next.

Can We Go Even Further With Symmetry?

We believe that we can make the Korz model conceptually more elegant by treating selectors as coordinates in a "selector" dimension. Selectors would then move into slot guards rather than being separate elements of slots. For example, a slot defined in Korz as

```
method {rcvr ≤ pointParent, device} display {...}
```

would instead be written as

```
method {rcvr ≤ pointParent, device, selector ≤ display} {...}
```

This would allow a slot that is to match all messages with selector `display` to be written as

```
method {selector ≤ display} {...}
```

Also, with selectors as coordinates, it would be possible to define parentage, supporting inheritance. This could be done explicitly, as with other kinds of coordinates, or possibly using pattern matching, where a pattern is considered an ancestor of every selector that it matches. For example, a slot that matches all setters using the standard naming convention could be written as

```
method {selector ≤ set*} {...}
```

The two extensions above have yet to be defined fully and implemented, but we believe that, rather than being special-purpose mechanisms, they would enhance the Korz model in an elegant and consistent manner. Korz should then be a suitable foundation for languages that deeply integrate the object-, aspect- and context-oriented paradigms (and possibly feature-oriented also; this needs exploration).

An even more tantalizing – and more speculative – extension is to devise a suitable way to shift perspective so that the slot's *contents* becomes one more component of its guard, rather than a separate contents component. Then the guard becomes the entire slot, possibly leading to a unification of the object and relational models.

We look forward to in-depth exploration of these and other issues relating to symmetric multidimensional models at the workshop, and to input from colleagues with deep insight from a number of different perspectives.

References

[Elra01] Elrad, T. et al. (eds.), 2001. Special section on aspect-oriented programming. *CACM*. 44, 10.

[Gabr91] Gabriel, R. et al. 1991. CLOS: Integrating object-oriented and functional programming. *CACM, 34, 9.*

[Hirs08] Hirschfeld, R. et al. 2008. Context-oriented Programming. *JOT*. 7, 3.

[HO93] Harrison, W. and Ossher, H. 1993. Subject-Oriented Programming: A Critique of Pure Objects. *OOPSLA'93.*

[Kicz97] Kiczales, G. et al. 1997. Aspect-Oriented Programming. *ECOOP'97*.

[Tarr99] Tarr, P. et al. 1999. N degrees of separation: multi-dimensional separation of concerns. *ICSE'99*.

[UA10] Ungar, D. and Adams, S. 2010. Harnessing Emergence for Manycore Programming: Early Experience Integrating Ensembles, Adverbs, and Object-based Inheritance. *OOPSLA'10 Short Paper.*

[Unga14] Ungar, D. et al. 2014. Korz: Simple, Symmetric, Subjective, Context-Oriented Programming. *Onward'14.*

[US87] Ungar, D. and Smith, R. B. 1987. Self: The Power of Simplicity. *OOPSLA'87.*