Korz: Simple, Symmetric, Subjective, Context-Oriented Programming

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

Abstract

Korz is a new computational model that provides for context-oriented programming by combining implicit arguments and multiple dispatch in a slot-based model. This synthesis 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 or aspects. With Korz, a system consists of a sea of method and data slots in a multidimensional space. There is no fixed organization of slots into objects – a slot *pertains* to a number of objects instead of being *contained* by a single object – and slots can come together according to the implicit context in any given situation, yielding subjective objects. There is no dominant decomposition, and no dimension holds sway over any other. IDE support is essential for managing complexity when working with the slot space and with subjectivity, allowing the task at hand to dictate what subspaces to isolate and what dominance of dimensions to use when presenting nested views to the user. We have implemented a prototype interpreter and IDE, and used it on several examples. This early experience has revealed much that needs to be done, but has also shown considerable promise. It seems that Korz's particular combination of concepts, each well-known from the past, is indeed more powerful than the sum of its parts.

Categories and Subject Descriptors D3.3 **[Programming Languages]:** Language Constructs and Features – Classes and objects

Keywords Subjectivity; Context; Multidimensionality; Programming

Onward! 2014, October 20 - 24 2014, Portland, OR, USA

http://dx.doi.org/10.1145/2661136.2661147

1. Introduction

Newtonian mechanics did a great job for terrestrial speeds, but when it turned out that light from a moving headlight traveled just as fast as light from a streetlight, a new physics was required to think effectively about fast-moving objects and electromagnetic radiation. Object-oriented programming does a great job for ontologies with a single dimension of variation, but creaks and groans when a second dimension enters the picture. The object-oriented programmer can easily model a system with one dimension of variation using inheritance, but when faced with a second dimension has to resort to the visitor pattern, strategy pattern, or an aspect-oriented methodology [Elra01]. Any of these can help, but only at the expense of weighing down the elegance of objects with additional concepts, and often at the expense of a potentially cumbersome and tricky refactoring of the code. Context-oriented programming offers a way out by adding implicit context to the state of a computation and using it to select from among behavioral variations [Hirs08]. The variations are usually reified as layers, however – an additional concept. Korz provides context-oriented programming, but by viewing the system as a uniform sea of slots, rather than objects and layers.

Just as Self reformulated the Smalltalk model of object-oriented computation in terms of a smaller, more-primitive set of concepts [US87], the Korz work described herein attempts to reformulate context-oriented computation in terms of a smaller, more primitive set of concepts that are simpler to work with yet more flexible and powerful. When moving from Smalltalk to Self, some of the language concepts, such as classes, became organizational patterns, such as traits. Likewise, when moving from some other context-oriented language to Korz, some of the language concepts, such as objects and layers, become organizational patterns.

With the Korz computational model, a system consists of a sea of *slots* (containing data values or methods), organized in a multidimensional *slot space*. Computation occurs in a *context*, which is also multidimensional, binding specific values to some or all of the dimensions in the slot space. At each computation step, a slot is selected from the space, using multiple dispatch that is based on the context, a selector, and explicit arguments, and then that slot is evaluated. The context is implicitly passed along to this evaluation, and hence serves as a set of implicit arguments.

Korz reduces to procedural programming in the zero-dimensional case, and object-oriented programming in the one-dimensional case (the single, implicit context element

being the "self" or "this" object). We believe Korz to be simpler, more flexible, more dynamic, and more expressive than previous approaches, particularly for evolving a program when additional kinds of variation are needed.

Multiple dispatch, implicit named arguments, and slot (or generic function) based models have each been around for a long time. Korz's contribution lies in their combination, which yields more than the sum of the parts. Multiple dispatch supports multiple dimensions of variation, implicit arguments support evolution and contextual programming, and the slot-based metaphor allows for subjective gathering of slots into different "objects" for different situations. Together, they allow a program to be easily extended to accommodate new kinds of variation and new perspectives.

We have built and exercised a prototype Korz implementation using the Self language, virtual machine and environment. Our Korz prototype includes an interpreter, debugger, and a partial interactive development environment (IDE). The syntax used in the prototype is based on Self syntax; however, to enhance the readability of examples in this paper, we present them in a Java(Script)-like syntax.

This paper describes the Korz model, discusses the principles behind it, and shows how it eases program construction and evolution. Section 2 gives an overview of Korz concepts, using a simple example. Section 3 provides a Korz language definition. Section 4 provides a more comprehensive Korz programming example, illustrating some of the power and benefits of the Korz approach. Section 5 discusses some issues arising out of the example, briefly illustrating our prototype Korz IDE. Section 6 discusses the programmer's conceptual model of a Korz system. Subsequent sections describe related and potential future work, and the appendices explore more deeply some interesting issues arising out of the present work.

2. Overview of Korz Concepts

This section introduces basic Korz concepts and terminology using a simple stack example shown in Figures 1 through 3. The stack implementation evolves to include a variant with assertion checking, and the particular collection of slots seen by any given caller depends on its particular implicit context. This brief example illustrates fundamental Korz mechanisms, rather than exemplifying realistic Korz usage; the example is too simple to actually warrant use of Korz, or other advanced formalisms, in actual practice. Section 4 will provide a more comprehensive example, after Section 3 defines the language.

Korz adopts the radical stance of altering the fundamental language-level notion of object. 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 simple stack example shown in Figures 1 through 3, examples of coordinates include: that referred to by the literal `0`, the contents of the constant slots `true` and `stackParent`, and the contents of the variable slot `stack1`.

```
// (coord for) prototype parent, with method slots
def {} stackParent = newCoord;
method { rcvr ≤ stackParent } pop() {
   sp = sp - 1;
   return contents[sp];
}
method { rcvr ≤ stackParent } push(x) {
   contents[sp] = x;
   sp = sp + 1;
}
method { rcvr ≤ stackParent } copy() {
   ...
}
// (coord for) prototype, with data slots
def {} stack = newCoord extending stackParent;
var { rcvr ≤ stack } sp = 0;
var { rcvr ≤ stack } contents = array.copy();
```

Figure 1. Simple stack example – stack definition.

```
method {} main() {
   var stack1 = stack.copy();
   var stack2 = stack.copy();

   stack1.push(100);
      // -same as- {rcvr: stack1}.push(100);

   stack2.push(200);
   var sum = stack1.pop() + stack1.pop(); // Oops!
}
main(); // Results in negative sp!
```

Figure 2. Simple stack example – stack usage.

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 2, `{rcvr: stack1}.push(100)` is an example of a message send. The context for the message send will include the coordinate `stack1` 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 `assertions` dimension: `true` or possibly `false`. In some circumstances (discussed in subsequent sections), syntactic sugar can reduce the code for this message send to `stack1.push(100)`.

The code in Figure 1 is structured using a pattern pioneered in Self, 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.

The code in Figure 1 begins by creating a coordinate for method slots that will be inherited by the stack prototype and by all stacks copied from that prototype. The coordinate is contained in the constant slot stackParent, and because the braces in the slot guard '{} stackParent', are empty, that slot is globally accessible; i.e., the stackParent slot is not constrained at all with respect to the contexts from which it is accessible, so it is accessible from any context. Pop, push, and copy methods are then defined, and placed in slots with corresponding selectors, with slot guards that indicate that they pertain to stackParent, which we say is "in the role of" rcvr, or "along the dimension" rcvr. That is, messages sent with a context that includes the coordinate stackParent in the rcvr dimension, or any coordinate descended from stackParent, as well as an appropriate selector and argument list, will match these methods (regardless of any additional dimension-coordinate pairs in the context). Dimensions other than rcvr can be used, but rcvr is analogous to the 'receiver' or 'this' object of object-oriented languages, and is thus familiar (see Appendix C.2 for discussion). In keeping with the JavaScript-like syntax, the examples include '()' after invocations of methods like pop and copy that have no explicit arguments, even though Korz unifies state and behavior so that even variables are subjective, and such empty parentheses are optional.

The code in Figure 1 then creates a coordinate for data slots that constitute the prototype stack. That coordinate is contained in the constant slot stack, with an empty slot guard. Finally, Figure 1 defines sp and contents data slots pertaining to stack in the rcvr dimension. The '0' used as the initial value of sp is a literal that denotes a coordinate representing the number zero.

Figure 2 contains a globally-accessible main() method, and its invocation. main() begins by copying the stack prototype twice, and storing the resulting new coordinates in the variable slots stack1 and stack2, which are local to the main() method. The code in Figure 2 then sends a message with a context that includes stack1 in the rcvr dimension. The message also includes the selector push, and the argument 100. The context, selector and explicit arguments of the message send are all used to find an appropriate slot in the slot space – the slot whose slot guard best matches the components of the message send (the dispatch algorithm is discussed in detail in the next section). That slot is then evaluated, and a coordinate is returned. Thus, the message send {rcvr: stack1}.push(100) results in the push method slot of Figure 1 being evaluated, because rcvr: stack1 matches rcvr ≤ stackParent from the slot guard by virtue of the fact that stack1 extends stackParent (it was created by copying stack), as well as the fact that the selectors are the same and the arguments (100) match the parameters (x).

In the body of an invoked method, a coordinate from the context can be accessed using its role (dimension) name. Thus, during the execution of the push method that arises out of the message send discussed above, rcvr is bound to the coordinate contained in stack1, just as x is bound to the coordinate represented by the literal 100 (standard parameter binding). The context thus constitutes a set of implicit arguments. By virtue of syntactic sugar involving the rcvr dimension being implicit, references such as sp and contents in fact access the data slots pertaining to stack1. E.g., sp = sp + 1 in this invocation of push is equivalent to {rcvr: stack1}.sp = {rcvr: stack1}.sp + 1.

The second push in Figure 2 proceeds similarly, but for stack2. The final line of the main() method erroneously pops stack1 twice; and popping an empty stack leads to a negative stack pointer and program failure.

Figure 3 shows code that could simply be added to the existing application to evolve it by defining an assertion-checking variant of the pop method. The slot guard for this method slot is: {rcvr ≤ stackParent, assertions ≤ true} pop(), which indicates that the method being defined is contained in a slot with selector pop, has no explicit positional parameters, and pertains to coordinate stackParent (along the rcvr dimension) and to coordinate true (along the assertions dimension); that is, the slot is constrained to only be accessible from contexts in which the coordinate in the rcvr dimension is stackParent, or a descendant thereof, and the coordinate in the assertions dimension is true, or a descendant thereof (and any other dimensions of the context are irrelevant to the accessibility of this slot). This slot guard *is more specific than* that of the pop method in Figure 1 because it has the extra assertions dimension, but is the same in other respects (specificity is defined in Section 3). The body of this method either detects a violation and issues an error message, or alters the context to instead include the coordinate false in the assertions dimension (or this could have been written to remove the the assertions dimension entirely) and then sends a pop() message, resulting in the original pop method in Figure 1 being invoked.

```
method { rcvr ≤ stackParent, assertions ≤ true }  pop() {
   if  ( sp ≤ 0 )
      error("Invariant violated: sp must be > 0");
   else
      return {assertions: false}.pop();
         // -or- return {-assertions}.pop();
}

{assertions: true}.main(); // Results in error msg
```

Figure 3. Code to add assertion checking.

The code at the end of Figure 3 invokes the existing main method, but now in the context assertions: true so that the assertion-checking variant of pop is used instead of the original pop method. This results in the attempted invariant violation being detected and prevented, and an error message being issued. The key to this example is that message sends for which the context does not include a coordinate for the assertions dimension, or includes the coordinate false for the assertions dimension, will see the original pop method; whereas message sends in which the context includes the coordinate true for the assertions dimension will see the assertion-checking pop method. Two facts are particularly noteworthy: (1) The binding assertions: true in the context of the send of the main message is implicitly carried through the main method; methods like main do not even need to be aware of new dimensions like assertions, and certainly don't need to deal with them in any explicit way.

(2) A pop message send with assertions: true in the context matches both pop slots, but the assertion-checking pop is used because it is more specific than the other.

This section concludes by contrasting Korz with existing (prototype-based) object-oriented languages such as Self and JavaScript. In such languages, the notion of object can be viewed as playing many roles: identity, context, reference, message destination, and set-of-slots. Computation proceeds by sending a message to (a reference to) an object, that reference often being the implicit context of the method enclosing the send, e.g., self or this. The message's target object determines a set of slots (including via inheritance). The message invokes the slot with the corresponding name, and that slot runs and returns another reference to an object. To create a new thing, one copies an old object to get a (reference to a) new object.

Korz deconstructs the notion of object, and recasts program structure within a multidimensional framework. Coordinate hierarchies and inheritance along each of a number of dimensions are supported. With Korz, the notion of a single object as the receiver of a message is replaced with a context for the message, consisting of zero or more dimension-coordinate pairs, which are used, together with the selector and arguments, to determine the slot to be evaluated. A given dimension-coordinate pair may be specified explicitly as part of the message send, or it may have been previously set and implicitly carried along a sequence of message sends / method invocations. A new thing can be created by expanding the slot space, using a copy method that creates a new coordinate and a number of slots pertaining to it, and returns the new coordinate.

3. Language Definition

A body of Korz code is termed a *slot space*: a collection of slots organized in a multidimensional space. Execution occurs when an expression is evaluated relative to the slot space. Expression evaluation usually involves sending messages. Each message send occurs in an implicit *context* (comprised of implicit arguments) and specifies a *selector* and explicit *arguments*. The context, selector and arguments (three kinds of *bindings*) are all used to find an appropriate slot in the slot space, by finding the slot whose *slot guard* (consisting of corresponding *constraints*) best matches the bindings. If a most-suitable slot is found, it is then evaluated to yield the result of the message send.

We first describe an abstract syntax for Korz slot spaces. We then describe the semantics of the interpreter. Both descriptions are semi-formal, with the intent of combining precision and readability.

3.1 Abstract Syntax

The abstract syntax described here should be thought of as the representation used by the interpreter rather than a representation close to any concrete syntax. It is illustrated using the simple stack example from Section 2, using the concrete syntax from that section.

3.1.1 Slot Space

A Korz *slot space* is a tuple $SS = (C, p, D, L, S)$ where:

- C is a set of *coordinates*,
- p is a *parent relation* on coordinates,
- D is a set of *dimension names,*
- L is a set of *selectors,*
- S is a set of *slots.*

Each *slot* consists of:

- A *slot guard,* $sg = (dcs, l, pct)$, where:
  - *dcs* is a *dimension constraint set*, made up of *dimension constraints* (which are *context*/*implicit parameter constraints*)
  - *l* is a *selector*
  - *pct* is a *parameter constraint tuple*, made up of *parameter constraints*
- *Contents*, which can be:
  - A coordinate, or
  - The special *assignment primitive,* or
  - A *method body*, which consists of:
    - 0 or more *local variable declarations,* and
    - An *expression,* usually a sequence of sub-expressions, which can be *message sends* or various other forms.

In this exposition, whenever we have tuples in the abstract syntax, we use the component names as the names of functions providing access to the components. Thus for *SS* above, *C*(*SS*) denotes the coordinate set of *SS*, *p*(*SS*) denotes its parent relation, etc.

Each of these elements, and their sub-elements, are now described in more detail. The exposition is done mostly bottom-up, so that we can keep building on known concepts; the map above puts the elements in context.

3.1.2 Coordinates

C is a set of *coordinates.* A coordinate $c \in C$ is an immutable value that serves as an *identity*. Examples of coordinates from Section 2 are: the literal 0, and the contents of the constant slot true and of the variable slot stack1.

Given two coordinates, one can determine whether or not they are in fact the same coordinate, i.e., whether $c_1 = c_2$. Some coordinates may be numbers, characters or strings, denoted by literals in the usual way. One special coordinate, denoted here by *any,* is always in *C*. *Any* is never explicitly written. We discuss *any* below. Other coordinates are created on demand by the *coordinate creation primitive* (denoted *newCoord* in the concrete syntax used in the stack example). This primitive is an expression but *not* a literal. Each time it is used, a new coordinate is created that is guaranteed not to be the same as any other coordinate.

Coordinates are analogous to object IDs/pointers in pure object-oriented languages (OOPLs): In an OOPL, every value refers to some object; in Korz, every value is some coordinate. In an OOPL, once an object is created, there may be no textual expression denoting it; in Korz, once a coordinate is created, there may be no textual expression denoting it. In an OOPL, we write object1 or aCar when writing about an object; the identifiers denote variables containing object references. In Korz we write coordinate1

or aCar when writing about a coordinate; the identifiers denote variables containing coordinates.

3.1.3 Parents and Ancestors

Korz supports inheritance through the parent relation on coordinates, $p: C \times C \rightarrow \{ T, F \}$, where T and F are the Boolean values. If $p(c_1, c_2) = T$, then c_2 is said to be a *parent* of c_1. In Section 2, stackParent is the parent of stack. By definition, *any* is the parent of exactly those coordinates that have no other parents.

The reflexive, transitive closure of the parent relation induces a partial order of generality/specificity:

$$c_1 \preccurlyeq c_2 \equiv c_1 = c_2 \vee p(c_1, c_2) \vee \exists\, c' \in C \text{ such that } c_1 \preccurlyeq c' \wedge c' \preccurlyeq c_2$$

The partial order relation '$\preccurlyeq$' can be read 'is at most as general as' or 'is at least as specific as,' and is analogous to the subtyping relations found in many languages. Like a subtyping relation, '$\preccurlyeq$' is partial because two coordinates may be unrelated by parentage, with neither being at least as specific as the other. However, every coordinate is at least as specific as *any*:

$$\forall\, c \in C: c \preccurlyeq any$$

The generality/specificity partial order relation will be extended to composite structures involving coordinates. In all cases, equality can be defined in the usual way:

$$x = y \equiv x \preccurlyeq y \wedge y \preccurlyeq x$$

3.1.4 Identifiers and Dimension Names

As is customary, an identifier is a sequence of a restricted set of characters. Examples of identifiers from Section 2 are: sp, pop, push and x. Identifiers can be compared for equality, and are used for variable and parameter names and the like. A Korz slot space includes a set, D, of identifiers used as dimension names, and hence defining the dimensional structure of the slot space. Examples of dimension names from Section 2 are: rcvr and assertions.

3.1.5 Selectors

L is a set of selectors. A selector is a sequence of characters used in a message to indicate the desired action, including method invocation, variable access, and variable assignment. Selectors can be compared for equality.

Korz selectors are just like those in other languages. In object-oriented languages in the C family, selectors are method names (identifiers), and sometimes operator symbols (C++). Smalltalk and Self use identifiers for unary selectors, operator symbols for binary operators and sequences of one or more colon-terminated identifiers for keyword selectors.

The precise syntax of Korz selectors is not material here. In the simple stack example we used identifiers for selectors. Examples from Section 2 are: pop and push.

3.1.6 Context: Dimension Binding Set

Each step of Korz execution consists of evaluating an expression in a *context*, which is a set of dimension bindings that are passed implicitly with invocations. A *dimension binding* is a pair, $db = (dim, coord)$, where $dim \in D$ is a dimension name, and $coord \in C$ is a coordinate. The coordinate specifies a *binding* for that dimension: a particular position on the dimension. A given coordinate can be used in more than one dimension. For example, one can imagine true being bound to a number of different dimensions.

A *dimension binding set*, $dbs = \{ db_1, db_2, \ldots, db_n \}$ is a set of 0 or more dimension bindings, containing at most one dimension binding per dimension of the slot space. Not all dimensions in the slot space need be mentioned in *dbs*; any dimension not mentioned is considered irrelevant. A context is a dimension binding set.

3.1.7 Argument Tuple

In addition to implicitly-passed arguments as the values of dimensions in the context, Korz supports explicitly-passed positional arguments. An *argument tuple*, $args = (arg_1, arg_2, \ldots, arg_n)$ is a tuple of 0 or more arguments, each argument being an expression (defined below).

3.1.8 Dimension Constraints

A slot space includes dimension constraints that will participate in slot selection (below) by constraining the set of acceptable coordinates that may be bound to a specific dimension. A *dimension constraint* is a pair, $dc = (dim, coord)$, where $dim \in D$ is a dimension name and $coord \in C$ is a coordinate. The coordinate specifies a particular position on the dimension. Examples of dimension constraints from Section 2 are: rcvr ≤ stack and assertions ≤ true .

A dimension constraint means that the coordinate bound to the dimension is constrained to be at least as specific as the coordinate specified in the constraint (i.e., the same as the coordinate in the constraint, or more specific than the coordinate in the constraint). Wherever a dimension constraint is needed, a dimension name alone may be written, omitting the coordinate. In that case the coordinate is understood to be *any*.

A dimension constraint can be tested to see if it is at least as specific as another dimension constraint:

$$dc \preccurlyeq dc' \equiv dim(dc) = dim(dc') \wedge coord(dc) \preccurlyeq coord(dc')$$

A *dimension constraint set*, $dcs = \{ dc_1, dc_2, \ldots, dc_n \}$, is a set of 0 or more dimension constraints, containing at most one dimension constraint per dimension of the slot space. When we get informal, we may use *context constraint* as a synonym for dimension constraint set.

A dimension binding (Section 3.1.6) can be tested to see if it satisfies ('$\sqsubseteq$')a dimension constraint:

$$db \sqsubseteq dc \equiv dim(db) = dim(dc) \wedge coord(db) \preccurlyeq coord(dc)$$

A dimension binding **set** satisfies a dimension constraint set if every constraint is satisfied by one of the bindings:

$$dbs \sqsubseteq dcs \equiv \forall\, dc \in dcs \;\; \exists\, db \in dbs \text{ such that } db \sqsubseteq dc$$

The binding set may include bindings for additional dimensions; since these dimensions are absent from the constraint set, they are considered unconstrained and hence irrelevant to this satisfaction relation.

A dimension constraint **set** is at least as specific as another dimension constraint set if it has extra dimensions or, in the case of equal dimensions, its coordinates are at least as specific. Inheritance in the matching dimensions does not matter in the case where one dimension constraint set has additional dimensions. In other words, additional dimensions trump inheritance, which is why line 3 below does not test for specificity:

$dcs \preccurlyeq dcs' \equiv$
1. $|dcs| > |dcs'| \wedge$
2. $\forall\, dc' \in dcs'\ \exists\, dc \in dcs$ such that
3. $dim(dc) = dim(dc')$
4. $\vee$
5. $|dcs| = |dcs'| \wedge$
6. $\forall\, dc' \in dcs'\ \exists\, dc \in dcs$ such that $dc \preccurlyeq dc'$

The implications of having additional dimensions trump inheritance are discussed further in Appendix B.

3.1.9 Parameter Constraints

A method slot that requires positional arguments will declare (and optionally constrain) the corresponding formal parameters with parameter constraints. A *parameter constraint* is a pair, $pc = (param\text{-}name, coord)$, where *param-name* is the parameter name, which is an identifier, and $coord \in C$ is a coordinate, which constrains the corresponding argument to be at least as specific as *coord*. *Coord* may be *any*, declaring but not constraining the parameter. In a concrete syntax, the coordinate is likely to be omitted in this unconstrained case, as was done in Section 2.

A parameter constraint is thus similar to a dimension constraint, but constrains the value of an argument rather than a dimension, and the declaration is used to declare an explicitly-passed formal parameter, rather than an implicitly-passed value for a dimension in the context.

A *parameter constraint tuple* $pct = (pc_1, pc_2, \ldots, pc_m)$ is an ordered tuple of 0 or more parameter constraints. The i^{th} parameter constraint of a parameter constraint tuple, *pct,* is denoted by $pc_i(pct)$. The arity of a parameter constraint tuple *pct*, denoted by $|pct|$, is the number of parameter constraints in the parameter constraint tuple. Informally, we may use 'parameter guard' for 'parameter constraint tuple'.

For example, the method below includes a single parameter constraint, named `x` :

```
method {} push( x ≤ number ); // x must be a number
```

Arguments may be tested for satisfaction against parameter constraints. An argument tuple satisfies a parameter constraint tuple if they have equal arity and every argument is at least as specific as the corresponding constraint:

$$args \sqsubseteq pct \equiv |args| = |pct| \wedge \forall\, i \in [1, |args|]:\ args_i \preccurlyeq coord(pc_i(pct))$$

Parameter constraint tuples may be compared for specificity. They require equal arity to be comparable, since methods with different numbers of parameters can never match the same message:

$pct \preccurlyeq pct' \equiv$
1. $|pct| = |pct'| \wedge$
2. $\forall\, i \in [1, |pct|]:$
3. $coord(pc_i(pct)) \preccurlyeq coord(pc_i(pct'))$

Since parameters are passed positionally, the *param-name*s play no part in specificity; as in many languages, they are used only to provide access to the argument values within the method body.

3.1.10 Slot Guards

A slot guard specifies the conditions for a slot to match a specific message, and hence be a candidate for evaluation in response to that message. The matching depends on three factors: the implicit *context* in which the message is sent (a dimension binding set), the *selector* used in the message, which indicates the desired action, and the explicit *arguments* (actual parameters) provided as part of the message. Accordingly, a *slot guard*, $sg = (dcs, l, pct)$, is a triple consisting of a dimension constraint set *dcs*, a selector $l \in L$ and a parameter constraint tuple *pct*.

Slot guards may be compared for specificity:

$$sg \preccurlyeq sg' \equiv dcs(sg) \preccurlyeq dcs(sg') \wedge l(sg) = l(sg') \wedge pct(sg) \preccurlyeq pct(sg')$$

Slot guards with different selectors are incomparable.

Examples of slot guards from Section 2 are:

```
{ rcvr ≤ stackParent } push(x)
```

and

```
{ rcvr ≤ stackParent, assertions ≤ true } pop().
```

3.1.11 Slot

A slot is a pair, $s = (sg, contents)$, where *sg* is a slot guard. No two slots in a slot space may have equal slot guards (i.e., slot guards all of whose components are equal, ignoring parameter names). *Contents* may be one of:

- A **coordinate**, in which case the slot is a *data slot*.
- The **assignment primitive**, in which case the slot is an *assignment slot.* In this case the parameter guard must specify a single parameter (to hold the value to be assigned), and the assignment slot must be paired with a data slot (thus forming a getter/setter pair). This pairing might be done using selector conventions, such as 'x' for a data slot and 'x:' or 'setX' for the corresponding assignment slot. A message sent to the assignment slot sets the value of the corresponding data slot.
- A **method body** (defined below), in which case the slot is a *method slot.*

Examples of slot declarations from Section 2 are:

```
var {rcvr ≤ stack} sp = 0;
```

and

```
method {rcvr ≤ stackParent} pop()  { ... } .
```

The `var` in this syntax declares both `sp` as a data slot and also a corresponding assignment slot that is invoked by assignment expressions like '`sp = 0`.' The `method` indicates that `pop` is a method slot.

3.1.12 Method Body

A *method body* is a pair (*vars*, *exp*), where *vars* is a sequence of 0 or more local variable declarations, and *exp* is an expression (usually consisting of a sequence of sub-expressions). A local variable declaration is a pair (*name*, *value*), where *name* is an identifier and *value* is an expression specifying the initial value. In a concrete syntax, the value can be omitted, and is then taken to be the literal nil.

3.1.13 Expression

An expression is a literal, the coordinate creation primitive, a message send, a block declaration, or a sequence of (sub-) expressions. A variable reference is written as an identifier, which is actually a parameterless message send whose effect is to return the coordinate contained in the variable. As in Self, this unification of variable access and message send is important to achieving unification of state and behavior. In Korz, it enables variable access and assignment to depend on context in just the same way as method invocation.

3.1.14 Dimension Modifier Set

When the need arises to execute a sub-expression with a different set of dimension bindings (i.e. in a different context) than is used for its enclosing expression, a *dimension modifier set* is used. A dimension modifier set, $dms = \{ dm_1, dm_2, \ldots, dm_n \}$ is a set of dimension modifiers, containing at most one dimension modifier per dimension of the slot space. A *dimension modifier* is a pair $dm = (dim, e)$, where $dim \in D$ is a dimension name and e is either an expression, which evaluates to a coordinate; or the symbol *'−'*, which indicates that any existing binding to the associated dimension should be removed. Examples of dimension modifier sets from Section 2 are: {assertions: true} and {-assertions}.

A dimension modifier contains an expression, which is evaluated when the modifier is used, whereas a dimension binding or dimension constraint contains a coordinate, which requires no evaluation.

3.1.15 Message Send

A message send is a triple $m = (dms, l, args)$, where *dms* is a *dimension modifier set*, $l \in L$ is a selector, and $args = (arg_1, arg_2, \ldots, arg_n)$ is an argument tuple. A message send is evaluated relative to a dimension binding set (i.e., in a context). The dimension modifier set serves to specify how the *incoming* dimension binding set (a.k.a. incoming context) should be modified to obtain the *evaluation* dimension binding set (a.k.a. evaluation context), which is used to find and evaluate the appropriate slot. Examples of message sends from Section 2 are:

```
{-assertions}.pop()
```

and

```
stack1.push(2)
```

which is syntactic sugar for

```
{rcvr: stack1}.push(2).
```

In the former example, regardless of any binding for assertions in the incoming dimension binding set, the evaluation dimension binding set would include no binding for assertions. Other bindings, e.g., of rcvr, are left unchanged.

3.1.16 Block Declaration

Blocks are interesting in Korz, and have been implemented in our prototype. They have much in common with blocks in Self, but must also deal with binding dimensions appropriately. Space precludes discussion of their details here.

3.2 Semantics

3.2.1 Execution

A Korz execution request is a triple (*dbs*, *e*, *SS*), where *dbs* is a dimension binding set (a.k.a. a context), *e* is an expression and *SS* is a slot space. Such a request corresponds to a top-level invocation, such as from a read-eval-print loop or IDE. In response to the request, the expression *e* is evaluated in the context of dimension binding set *dbs*, using *SS* to find any slots involved in the evaluation. The result of the execution is the value of the expression *e*.

3.2.2 Expression Evaluation

Evaluation of an expression returns a value, which is a coordinate. Since it always occurs relative to a dimension binding set (i.e., in a context) and uses slots in a slot space, expression evaluation is defined by the function val(*dbs*, *e*, *SS*), where *dbs* is a dimension binding set, *e* an expression, and *SS* a slot space.

- If *e* is a **literal**, val(*dbs, e, SS*) is the value of the coordinate denoted by the literal; *dbs* and *SS* are irrelevant.
- If *e* is the **coordinate creation primitive,** val(*dbs, e, SS*) is a new, unique coordinate, which is added to *C*(*SS*) as a side-effect; the dimension binding set is irrelevant.
- If *e* is a **sequence of expressions,** $(e_1, e_2, \ldots, e_n)$, each expression is evaluated in sequence: v_i = val(*dbs,* e_i*, SS*). The value of the expression as a whole is the value of the last one, v_n.
- If *e* is a **message send** *m* = (*dms, l, args*), then evaluation involves finding an appropriate slot in *SS*, and evaluating it, which is a four-step process:
  val(*dbs,* (*dms, l, args*)*, SS*) =
  1. *args′* = val(*dbs, args, SS*);
  2. *dbs′* = modifyDimBindings(*dbs, dms, SS*);
  3. *s* = lookup(*dbs′, l, args′, SS*);
  4. **return** val(*dbs′*, *args′*, *s*, *SS*)

  Step 1 evaluates each argument expression to produce a tuple of coordinates. Step 2 applies the dimension binding modifier to the incoming dimension binding set to obtain the evaluation dimension binding set (concepts which were introduced above). Step 3 finds the unique slot to evaluate; this might fail, in which case an error occurs. The error can be handled in various ways, such as bringing up the debugger. Step 4 evaluates the slot, and returns the result as the value of the message send.

As mentioned earlier, details of blocks are not described.

3.2.3 Dimension Binding Modification

modifyDimBindings(*dbs, dms, SS*) =
1. *dbs′ = dbs;*
2. **for each** dimension modifier *dm* in *dms*:
3. **if** ∃ *db* ∈ *dbs′* such that *dim(db) = dim(dm)*,
4. remove *db* from *dbs′*;
5. **if** *e(dm)* is not '−',
6. add *db* = (*dim(dm)*, val(*dbs*, *e(dm)*, *SS*)) to *dbs′*;
7. **return** *dbs′*

3.2.4 Slot Lookup

Slot lookup involves attempting to find a single, most specific slot whose guard matches the message:

lookup(*dbs, l, args, SS*) =
1. m = { *s* ∈ *S*(*SS*) | matches(*sg*(*s*), *dbs*, *l, args, SS*) };
2. removeLessSpecific(*m*, *SS*);
3. **if** | m | = 1, **return** the member of m;
4. **if** | m | = 0, **error** 'Not understood';
5. **if** | m | > 1, **error** 'Ambiguous'

3.2.5 Slot Guard Matching

Slot guard matching requires matching of the selectors, and satisfaction of the constraints:

matches(*sg, dbs*, *l, args, SS*) ≡
1. *dbs* ⊑ *dcs*(sg) ∧
2. *l* = *l* (*sg*) ∧
3. *args* ⊑ *pcs*(sg)

3.2.6 Slot Specificity

Once all matching slots have been found, we need to remove any that are less specific than other matching slots:

removeLessSpecific(*m*, *SS*) =
1. **for each** slot *s* ∈ *m*:
2. **if** ∃ s′ ≠ s ∈ *m* such that *sg*(*s*) ≼ *sg*(*s′*),
3. remove *s′* from *m*

It is safe to use '≼' without worrying about equality between slot guards because of the restriction that no two slots in *SS* can have equal slot guards.

3.2.7 Slot Evaluation

If the contents of a slot is a coordinate, that is its value.

If the contents of a slot is the assignment primitive, the contents of the corresponding data slot is replaced with the value of the first and only argument, arg_1, and the new contents is the returned value.

If the contents of a slot is a method body, then an *activation* is created to constrain the scope of local variables, the dimension binding set (context) is updated to record the activation, and then the expression is evaluated:

val(*dbs′, args′, s, SS*) =
1. *a* = createActivation(*dbs′, args′, s*);
2. *dbs″* = activation-dbs(a, *dbs′*);
3. **return** val(dbs″, *expression*(*contents*(*s*)), *SS*)

The createActivation function creates a coordinate to represent the activation, and creates data slots associated with it in the 'activation' dimension to enable variable access. For each formal parameter, *p,* in the slot guard, a (constant) data slot is created with selector *param-name(p)* and value the positionally-corresponding argument value. For each dimension constraint in the slot guard, a (constant) data slot is created with selector the dimension name and value taken from the dimension binding for that dimension. This slot enables the value bound to a dimension in the context to be accessed in the method body using the dimension name. For each local variable declared in the method body, a data slot pair (including an assignment slot) is created with selector the variable name and value the value of the initialization expression. The slot guards for all these data slots include all of the dimension constraints for the method slot plus a constraint pairing the activation dimension with the activation coordinate.

In the activation-dbs function, the dimension binding set, *dbs′*, is then enhanced with a binding of the activation coordinate to the 'activation' dimension.

Finally, the expression in the method body is evaluated to yield the value of the slot. This evaluation occurs with the 'activation' dimension bound to the activation coordinate above, so messages whose selectors are parameter, dimension or local variable names will access slots set up by createActivation, yielding the expected results.

3.2.8 Cloning

Since Korz is based on prototypes instead of classes it creates new things by cloning rather than instantiation. Korz provides the *clone primitive* for this purpose, invoked via a message send. The semantics of clone are:

clone(*c*) =
1. *c′* = a new coordinate via coordinate-creation primitive
2. **for each** *s* ∈ *SS* such that *sg*(*s*) contains *c*:
   s′ = a new copy of *s*, with *c′* replacing *c* in *sg*(*s′*)
3. **add** *s′* to *SS*

4. Example

We now present a fuller example that illustrates how Korz can support context dependence, evolution, symmetry and subjectivity. The example is inspired by the colored-point example that was a popular vehicle for discussing evolution of object-oriented programs. Except for the actual display of pixels on screens, the code in the figures below has been written and tested on our prototype Korz interpreter and IDE. To avoid confusion, the discussion below uses *coordinate* only in its Korz language sense, and will use *position* to talk about where things display on a screen.

4.1 Make a point

The starting point for this example is a cartesian, colored point. First the program defines Korz coordinates for the prototypical point (from which new points will be cloned) and its parent (with which methods applicable to all points will be associated).

To make the coordinates accessible, the program defines them as the contents of slots. The code below therefore cre-

ates two slots, each containing a new coordinate. The point coordinate is declared to have the pointParent coordinate as its parent:

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

The empty slot guard "{}" for point means that, in any context, the point message will result in evaluation of the slot just defined and return of the point coordinate (provided point is not overridden).

Now the program creates three assignable data slots associated with the prototypical point coordinate in the rcvr dimension (in other words, the implicit argument named rcvr; our reasons for using this particular dimension are discussed in Appendix C.2.):

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

The slot guards, {rcvr ≤ point}, specify that the slots are accessible only in contexts in which the rcvr dimension is bound to a point, i.e., to a coordinate that is at least as specific as the point coordinate. For example, the message {rcvr: point}.color (which can be sugared as point.color) will return the prototypical point's color.

This example needs a method to make a point:

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

The point.copy method creates a new coordinate whose parent is the same as point's, i.e., pointParent. (Recall that copy and copy() are equivalent.)

4.2 Add a method to display a point

This method declaration, consisting of slot guardand method body, defines a method that displays a point:

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

Its guard mentions the device dimension without specifying a coordinate. This construct has two related effects:

- First, any dimension mentioned in the guard must be present in the message context in order for the slot to be found. In this case, if there is no such dimension in the message context, this display slot will not be found.
- Second, as defined in Section 3.2.7, every dimension occurring in a method slot's guard is placed into the scope of the method, so that the dimension identifier can be used within the method as a reference to that dimension's coordinate from the incoming context.

Consequently, if the message context has a dimension named device, this slot can be found, and the coordinate that is bound to the device dimension becomes bound to 'device' and will be used for displaying.

4.3 Add a screen object that can draw pixels

The example needs a screen object to use as the device coordinate. The screen needs a drawPixel method:

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

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

Given these definitions, one would normally have constrained the device dimension in the display method earlier to be at least as specific as screenParent (or, more likely, a more general deviceParent coordinate). We left it unconstrained in this example to illustrate that capability.

4.4 Drawing a point and more complicated figures

If p1 is a point, and s is a screen (a coordinate at least as specific as screenParent), the programmer can write:

```
{rcvr: p1, device: s}.display
```

That was a significant amount of setup for a simple example, but we are now well positioned for more complex cases and for evolution.

Consider figures that contain many (e.g. three) points:

```
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
}
```

If f1 is such a figure and s is a screen, this will work: {rcvr:f1, device:s}.display. The screen dimension is passed down through the figure display method. That method doesn't have to care what device is being used. This is analogous to COP [Hirs08].

4.5 Add a dimension

Now suppose that it is necessary to extend this code to accommodate colorblind people. Realizing that colorblindness is a separate dimension from the figure or the device (it might apply to any figure and any device), the programmer need only define a more specialized drawPixel method slot to be used when a new isColorblind dimension is present and bound to true:

```
method { rcvr ≤ screenParent, isColorblind ≤ true }
drawPixel(x, y, c) {
  {isColorblind: false}
    .drawPixel(x, y, c.mapToGrayScale)
}
```

This code reuses the existing true coordinate as a coordinate in this new isColorblind dimension; such reuse is perfectly acceptable because the same coordinate can be used in different dimensions. Whenever drawPixel is sent to a context with a screen for the receiver and isColorblind bound to true, the new method will run instead of the old drawPixel method. It will map the color to grayscale, and then call the old method. This call invokes the old method because the original drawPixel slot's guard omitted isColorblind, which meant that it did not care about it, and would accept any value. The code could have used {-isColorblind} instead of {isColorblind: false} to remove the binding of that dimension from the context instead of rebinding it to false.

The programmer can test the new capability by evaluating:

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

In a complete system, isColorblind would probably not be included explicitly in the context of this message that calls display; isColorblind would probably have been added to the context somewhere up the call stack, such as when a colorblind user logs in, and then carried implicitly to this point in the code and be included implicitly in the context of this message.

4.6 Add another dimension

Now suppose that another (somewhat contrived) requirement comes up: the need to flip figures upside-down for Australian users. This requirement is an example of another common dimension, location, which can be a key factor in mobile applications. To satisfy this requirement, the programmer could introduce another dimension with boolean coordinates, such as isAussie, but it is better to generalize somewhat and introduce a few coordinates to be used in a new location dimension:

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

Now the code uses the location dimension in the guard of a new drawPixel slot, and applies the upside-down requirement to the entire southern hemisphere:

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

This new code can be tested with:

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

and the y coordinate is negated just as desired.

Now suppose one tests support for colorblind Australians:

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

This test fails, with an ambiguous error because two drawPixel slots match the message: the one for Australians and the colorblind one, and neither is more specific than the other. The fix is easy: a special-purpose slot whose guard is more specific than both of the existing guards and that specifies how these cases are to be combined:

```
method {
  rcvr ≤ screenParent,
  isColorblind ≤ true,
  location ≤ southernHemi
}
drawPixel(x, y, c) {
  {-isColorblind}.drawPixel(x, y, c.mapToGrayScale);
}
```

4.7 Add a specialization

As a final evolution example, consider specializing the display for Antarctica; since it is so cold that people wear goggles that may fog up, the image must be magnified by a factor of 2, in addition to being inverted:

```
method { rcvr ≤ screenParent, location ≤ antarctica }
drawPixel(x, y, c) {
  {-location}.drawPixel(2 * x, -2 * y, c);
}
```

Then

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

ends up invoking this method. The message

```
{ rcvr: f1, device: s,
  isColorblind: true, location: antarctica }.display
```

illustrates the implications of our decision to allow matches in extra dimensions to trump the specificity of the matches. See Appendix B for further details.

5 Issues Raised by Multidimensional Implicit Context

The move from single-dispatch object-oriented programming to a multidimensional contextual paradigm raises some interesting issues. Many of these will require further research. Appendix C contains more material on this topic.

5.1 Symmetry and Subjectivity

The issues of symmetry and subjectivity in Korz can be illustrated with portions of the code for the example in Section 4 as shown in the Korz prototype IDE.

In Figure 4, the green box at the top shows the slot

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

The ellipses near the right of the slot represent the method body, which can be viewed in detail by clicking the square button beside them. In standard object-oriented languages, this slot would be considered as belonging to screenParent. Indeed, it does belong to screenParent in Korz too (in the rcvr dimension), as shown in the blue box below and to the left of it. But it equally-well belongs to southernHemi (in the location dimension), as shown in the blue box below

and to the right of it. Whereas the green box shows an individual slot, each blue box shows a collection of slots that share a coordinate in one dimension, and can be regarded as constituting an *object* in Korz. Each slot in each object shows the other dimensions of variation. The buttons provided by the IDE for the coordinates in those dimensions allow navigation to views of those coordinates as objects. For example, pressing one of the buttons labeled '**location => southernHemi**' on the left of Figure 4 would show a view of **southernHemi** as an object, as at the middle right of Figure 4 (the prototype IDE uses double arrows in dimension bindings, rather than '≤' as used in this paper).

Thus, instead of an asymmetric organization with a dominant decomposition, in which one of the dimensions (e.g., object or class) is primary and the others (e.g., layers, aspects or subjects) are secondary, Korz's conceptual economy (i.e. no objects, no layers, just slots and coordinates) provides a *symmetric* organization, in which slots can be grouped into objects based on any dimensions. This notion of object is *subjective*. It allows slots to be gathered together into objects in ways that provide different abstractions or views, useful for different purposes. This applies to both data slots and method slots. A data slot modified via one view will manifest the new value in other views also.

6. Programming with Korz

In this section, we move from language and dispatch details to consider how programmers think about their Korz programs. Appendix D provides more depth on these topics.

Slot space versus object model: Traditional object-oriented programmers, when wanting an overall understanding of a program, think in terms of an *object model,* in which the inheritance hierarchy plays a key role in organization and overall understanding. In Korz, the multidimensional slot space assumes this role. Multidimensional spaces are conceptually simple and regular, but quickly become large and hence complex in detail. Sophisticated IDE support is critical to working with them effectively. Since many object-oriented programs actually deal with multiple dimensions of variation, Korz's paradigm, along with a suitable environment, may well actually ease the task of working with such programs.

Modularity: Dimensions provide a flexible and powerful modularization mechanism that can be used for program organization and presentation. A module can be represented by a specific dimension, or a coordinate within a specific dimension. However, the global scope of dimension names in Korz could present problems when merging two Korz slot spaces that have some dimension names in common if those names are used with different meanings in the two spaces.

Static analysis and programmer assistance: The dimensions and coordinates in the slot space provide valuable structural information to programmers, and can be used by an IDE for intelligent code completion in slot guards and dimension binding modifiers. Though Korz is not statically typed, the constraints in slot guards provide a good deal of information that might be used for type inferencing in the same fashion as Agesen's work for Self [AU94], and hence for intelligent code completion as well.

7. Previous Work

7.1 Implicit Arguments

The utility of implicit arguments (or dynamic scoping) for evolving programs to take additional aspects of context into account has a long history [HP01, Lewi00]. These efforts did not link implicit arguments to dynamic dispatch, though. Such linkage is possible in CLOS (see below).

7.2 Multiple Dispatch

Likewise, multiple dispatch is nothing new, including the integration with object-oriented programming [BG93, Cham92, Gabr91]. The **Flavors** paper explicitly included the idea of dispatching on two arguments [Moon86, page 4]. The **Julia** language dispatches methods based on multiple, positional arguments, though not on the keyword parameters [Beza14]. It does not include implicit arguments as Korz does.

Heinlein has experimented with mulitple-dispatch **Virtual Functions** in C++ [Hein05], but his dispatch mecha-

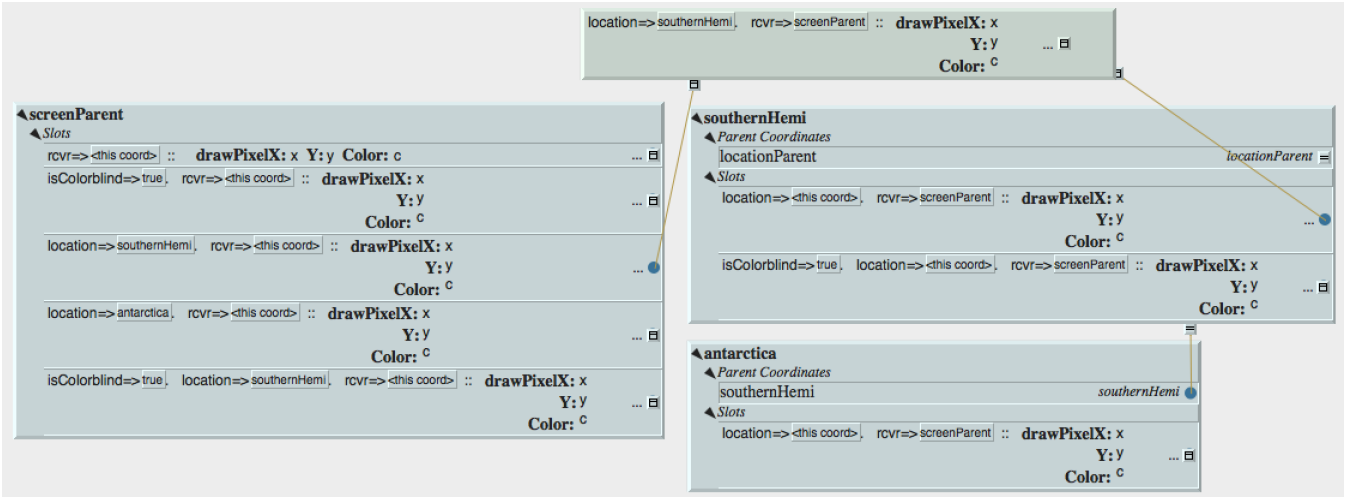

Figure 4: IDE views of a slot and the 'objects' it pertains to

nism chose the most-recently-executed definition, not the most specific. Pirkelbauer et al have presented **Open Multi-Methods** for C++ and have shown them to be efficient [Pirk07].

Delegation Layers [Oste02] is another close relative to Korz: combining delegation with virtual classes, but retaining the object as a fundamental entity.

Traits [Schä03] composed classes out of smaller units. As with multiple dispatch, the resulting composite behavior depends on multiple, independent factors. Unlike multiple dispatch, this idea uses static composition, and remains firmly in the object-oriented paradigm of a single runtime entity determining the dispatched method.

7.3 Static Multidimensional Context

Subject-oriented programming **(SOP)** [HO93] pioneered the notion of subjective objects, and modules called *subjects* were the precursors of COP layers. The original SOP paper [HO93] described dynamic activation and de-activation of subjects during execution, much like COP layers. However, implementations of SOP all performed subject-composition prior to execution. The same is true of most aspected-oriented [Kicz97] approaches, in which *aspects* are woven into classes before execution.

SOP was extended with multidimensional structure in multidimensional separation of concerns **(MDSOC)** [Tarr99]. Whatever modules were used in a program, e.g., classes, or subjects, slots were considered to be arranged in a multidimensional space, like that of Korz. MDSOC implementations performed composition of collections of slots into executable Java programs before execution. Korz adds dynamicity and conceptual simplicity by embracing the multidimensional space as the runtime program representation, and using multiple dispatch rather than a separate composition step. However, as of now, it does not support the richness of composition operators. Further details are provided in Appendix D.

7.4 Dynamic Multidimensional Context

A modified form of **Slate** [SA05] provided multiple dispatch in a Self-like, prototype-based setting, and even provides a subjective dispatch with one additional implicit argument. It does not appear to have been generalized to the extent of Korz, though.

With the notable exceptions of **Us** [SU96] and **PyContext** [Löwi07], as far as we know multiple dispatch has not usually been combined with implicit arguments. However, there is a tantalizing hint that the CLOS creators were going in that direction: *"it might be useful to use exogenous features for method selection. ... one might include the process in which the operation is being done"* [Gabr91]. **PyContext** [Löwi07] is notable because it does activate layers according to implicit arguments, and those layers can contain variables. It is one of the closest existing proposals to Korz, differing principally in the layers vs. sea-of-slots formulations. **ContextJ** [Hirs08] includes a "with" construct that may yield a similar effect.

Korz follows in the grand tradition of **Context-Oriented Programming (COP)** [Appe09, Hirs08, Löwi07], but departs from the earlier notion of **layers.** (Hirschfeld et al's paper [Hirs08] has an especially good treatment of prior work in this regard.) Earlier COP papers conceptualized the context-dependent information as a layer of (usually behavior, but state as well for **Us** and **PyContext**) slots overlaying a more universal object. When more than one dimension of variability is required, the layer formulation requires a concept of linearization, in order to select from competing layers. Korz departs from this tradition by viewing the system as a sea of slots, rather than objects and layers, more in the tradition of **CLOS** and **Cecil** [BG93, Cham92, Gabr91]. Our multidimensional formulation may be more parsimonious, in that it has only slots, not objects and layers. However, we have not yet addressed the "super" or "call-next-method" issue, which is solved by the layer linearization facility in these other languages. We believe the essential contribution of COP is *not* the ability to refer to layers at runtime, but rather the ability to vary behavior according to multiple dimensions at runtime.

Also, one advantage cited for COP is that layers are first-class entities that can be shown directly in source code. For Korz, we would rely on the IDE to group slots as needed to do the same thing. By not reifying groups of slots (layers) at the fundamental language level, we allow the IDE to present different groupings as they are needed by the user. Korz, **Us**, and **PyContext** all allow state (i.e. data slots) to be context-dependent, however only Korz goes so far as to consider object identity to be subjective. Thus, we avoid the dilemma of choosing layer-in-class vs class-in-layer, as defined by the JOT paper [Hirs08]. In a way, atomizing objects into slots, rather than retaining objects and adding layers, is analogous to the simplification made by the original Self paper: That paper showed how objects could play the roles of both instance and class; this paper shows how subjective collections of slots with multidimensional guards and implicit arguments can play the role of both objects and layers. In each case, IDE support restores the higher-level abstractions, while, we believe,the simpler underlying model provides for greater malleability.

In this paper we use *multidimensionality* in a slightly different sense than does Hirschfeld et al [Hirs08]: in our formulation, since we don't yet include the selector as a dimension, zero dimensions equates to functions, one dimension equates to object-oriented programming, and more than one dimension equates to dispatch by multiple implicit parameters. To us, this seems more consistent (except maybe for the selector bit) than their taxonomy in which first the selector, then the receiver, then the sender, then other parts of context are added to the dimensionality. Because object-oriented programming code typically uses small methods to get the most out of inheritance and polymorphism, we believe that using the sender explicitly is less useful than using a named implicit argument; there are likely to be many intervening methods between the "sender" of interest and the behavioral inflection point.

The idea of multidimensional implicit context was also developed in the **Lucid** community [AW77, FJ93, PP96, SG02, Wan05]. Both that work and ours were independently inspired by the role of implicit context in human communication. As in Korz, this work adds user-defined dimen-

sions to expressions, with the coordinates determined by implicit context. Just as Korz includes dimension modifiers, this intensional programming work includes intensional operators to navigate the context space. In Korz, context propagates down the call chain; in at least some of this work ([SG02]) it propagates along a data graph. Korz is an imperative object-oriented language, while Lucid and its descendants tend to employ variables and code and represent state as a time-series of immutable values.

There has even been a meld with Java [Wu09]. In this OOIP system, Java classes may have Lucid members, and Lucid expressions may refer to members of Java classes. Korz integrates multidimensional context into an object-oriented paradigm; in OOIP multidimensional context is tied to a Lucid paradigm, although Lucid code can be mixed with Java code.

Overall, the key distinguishing feature of Korz is that it addresses the challenging conjunction of graceful evolution and multidimensional variability with its small kernel of concepts, without the need for a variety of module types and their attendant operations, such as layer activation, composition or weaving. We believe that the combination of multidimensional dispatch and implicit context provides a small, elegant and yet powerful basis for building and, especially, evolving systems.

8. Future Work

To develop Korz further, we need to come up with a better syntax, code up more examples in it, and figure out what to do about **super.** We also want to push the slot name (selector) into just another dimension with, say, specificity (inheritance) based on pattern matching. With some way to limit access to a slot, such as lexical scoping, a coordinate contained in such a slot can be used to guard "private" slots, providing a more flexible mechanism to enforce invariants, even cross-cutting ones, than other schemes we are aware of. We have also started to experiment with dimensions that alter the behavior of the interpreter, such as handling failure or ambiguity. This is the key to supporting method composition/combination. For example Korz could support **Ensembles** [UA10] with a dimension that said: "run every slot for this message." But there is an even more tantalizing prospect: if we can 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.

The sea-of-slots, subjective object model of Korz poses an interesting challenge for its environment, which will have to tame its complexity. We believe it can do so by offering progressive disclosure of dimensions; supporting whatever view of the slot space is best suited for the task at hand, be it symmetric, a slice, or a projection; and by providing the illusion of objects in a given perspective. Such a perspective must be salient enough to be clear to a programmer without being constantly distracting.

9. Conclusion

Looking at programming through the lens of context, and wishing to make it easier to build systems whose behavior depends on multiple dimensions of context, we have proposed and implemented a prototype system for Korz, a new programming paradigm. Korz generalizes object-oriented programming from one (usually implicit) receiver to many (usually implicit) "receivers," each playing a named role when a slot is looked up for a message send. Consequently, rather than a Korz system being a collection of objects, at the most fundamental level it consists of a collection of slots, each bound to some set of coordinates. Each slot, rather than being included in an object, is linked to one or more coordinates (each assigned to a named role or dimension), by its slot guard. Each coordinate, rather than containing slots, is merely a point of identity with links to the related slots in appropriate roles. When a message is sent, it has a context consisting of a set of coordinates, each in a named role. The dispatch mechanism selects the most specific slot and runs it.

Although Korz's multidimensional collection of slots captures more organizational information than a unidimensional set of objects, it can often be helpful to view a system as a simple set of objects. If one freezes all but one dimension, thus allowing only one dimension to vary, the collection of slots takes on the structure of an object-oriented system, isomorphic to objects containing slots. However, rebinding one of the frozen dimensions to a different coordinate will change the contents and identities of the "objects" in the system. At this level, the system is subjective and relative to a given perspective.

Often when programming in an object-oriented style, the programmer creates an inheritance hierarchy along some dimension of variation, only to get stuck when required to add a second, orthogonal dimension of variation. Object-oriented practice dictates that the classes or prototypes then be split into a pluggable architecture, so that items varying along the second dimension can be plugged into items varying along the first. But this transformation is often difficult. Aspect-oriented, feature-oriented and related approaches address this problem through additional modularity constructs to encapsulate concerns that do not align with the dominant dimension, but this complicates the object model with these additional kinds of modules and the means to compose or weave them. Korz offers a better way: additional slots and dimensions can be added incrementally, using only the core mechanism of Korz.

Korz's contribution lies in combining a relatively small number of pre-existing concepts: multiple dispatch, implicit, symmetric, named arguments, and slots with unified state and behavior as the fundamental particle. This combination yields more than the sum of the parts. Multiple dispatch supports multiple dimensions of variation, implicit arguments support evolution and contextual programming, and the slot-based metaphor allows for subjective gathering of slots into different "objects" for different situations. Together, they allow a program to be easily extended to accommodate new kinds of variation and new perspectives.

Acknowledgments

Sam Adams provided the impetus for us to explore contextual programming, and helped us fit this work into our organization. We are also very grateful to Mark Wegman and IBM Research, and to Suparna Bhattacharya and Sam Adams for helping us to prepare this paper.

References

[AU94] Agesen, O. and Ungar, D. 1994. Sifting Out the Gold. *OOPSLA'94.*

[Appe09] Appeltauer, M. et al. 2009. A Comparison of Context-oriented Programming Languages. *COP'09.*

[AW77] Ashcroft, E. A. and Wadge, W. W. Lucid, a Nonprocedural Language with Iteration. 1977. *CACM*. 20, 7.

[Bato94] Batory, D. et al. 1994. The GenVoca Model of Software-System Generators. *IEEE Software.* 11, 5.

[Beza14] Bezanson, J. et al. 2014. The Julia Manual. *http://docs.julialang.org/en/release-0.2/manual/* accessed 7/18/14.

[BG93] Bobrow, D. G. and Gabriel, R. P. 1993. CLOS in Context: The Shape of the Design Space. *CACM*. May, 1993.

[Cham92] Chambers, C. 1992. Object-Oriented Multi-Methods in Cecil. *ECOOP'92.*

[Chun05a] Chung, W. et al. 2005. The Concern Manipulation Environment. *ICSE'05.*

[Chun05b] Chung, W. et al. 2005. Working with Implicit Concerns in the Concern Manipulation Environment. *AOSD'05 Workshop on LATE.*

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

[FJ93] Faustini, A. A. and Jagannathan, R. 1993. Multidimensional Problem Solving in Lucid. *SRI-CSL-93-03*. SRI International.

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

[Harr05] Harrison, W. et al. 2005. Supporting aspect-oriented software development with the Concern Manipulation Environment. *IBM Systems Journal.* 44, 2.

[Hein05] Heinlein, C. 2005. Global and Local Virtual Functions in C++. *JOT*. 4, 10.

[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.*

[HP01] Hanson, D. R. and Proegsting, T. A. 2001. Dynamic Variables. *PLDI'01.*

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

[Kicz01] Kiczales, G. et al. 2001. An Overview of AspectJ. *ECOOP'01.*

[Löwi07] Löwis, M. et al. 2007. Context-Oriented Programming: Beyond Layers. *ICDL'07.*

[Moon86] Moon, D. 1986. Object-Oriented Programming with Flavors. *OOPSLA'86.*

[Lewi00] Lewis, J. R. et al. Implicit Parameters: Dynamic Scoping with Static Types. *POPL'00.*

[Ossh96] Ossher, H. et al. 1996. Specifying subject-oriented composition. *TAPOS*, 2, 3.

[Oste02] Ostermann, K. 2002. Dynamically Composable Collaborations with Delegation Layers. *ECOOP'02.*

[OT00] Ossher, H. and Tarr, P. 2000. Hyper/J: multi-dimensional separation of concerns for Java. *ICSE'00.*

[Pirk07] Pirkelbauer, P. et al. 2007. Open Multi-Methods for C++. *GIPCE'07.*

[PP96] Plaice, J. and Paquet, J. 1996. Introduction to intensional programming. *Intensional Programming I.* World Scientific.

[Preh97] Prehofer, C. 1997. Feature-oriented programming: A fresh look at objects. *ECOOP'97.*

[SA05] Salzman, L. and Aldrich, J. 2005. Prototypes with multiple dispatch: An expressive and dynamic object model. *ECOOP'05.*

[Schä03] Schärli, N. et al. Traits: Composable Units of Behavior. *CSE 02-012,* Dept. of Computer Science and Engineering, Oregon Health & Science University.

[SG02] Stavrakas, Y. and Gergatsoulis, M. 2002. Multidimensional Semistructured Data: Representing Context-Dependent Information on the Web. *CAISE'02.*

[SU96] Smith, R. B., Ungar, D. 1996. A Simple and Unifying Approach to Subjective Objects. *TAPOS,* 18, 4.

[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.*

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

[Wan05] Wan, K. et al. 2005. A Context Theory for Intensional Programming. *CRR'05 Workshop on Context Representation and Reasoning.*

[Wu09] Wu, A. et al. 2009. Object-Oriented Intensional Programming: Intensional Classes Using Java and Lucid. *Software Engineering Research & Applications'10.*

Appendix A: The Name

The name "Korz" comes from Korzybski, whose Science and Sanity (Korzybski 1933) explained how much one's perspective influences one's perceptions and thinking.

Appendix B: Slot Lookup Specificity

The example in section 4.7 sheds light on an interesting design issue for multidimensional languages such as Korz:

```
method { rcvr ≤ screenParent, location ≤ antarctica } (1)
drawPixel(x, y, c) {
  { -location }.drawPixel(2 * x, -2 * y, c)
}
```

Consider the following case of message lookup, which was mentioned there, but not discussed:

```
{ rcvr: f1, device: s, isColorblind: true, location: antarctica
}.display
```

This will cause a message send with selector drawPixel and context

```
{ rcvr: s, isColorblind: true, location: antarctica }
```

The slot guard of the drawPixel method introduced in section 4.7 (and repeated above as (1)) is the most specific in the location dimension, but it does not mention the isColorblind dimension. On the other hand, the drawPixel method introduced at the end of section 4.6 has a guard constraining all three dimensions, but the constraint on location is looser (less specific):

```
method {                                                (2)
  rcvr ≤ screenParent,
  isColorblind ≤ true,
  location ≤ southernHemi
}
drawPixel(x, y, c) {
  {-isColorblind}.drawPixel(x, y, c.mapToGrayScale);
}
```

The rule that additional dimensions trump inheritance, defined in Section 3.1.8, means that method (2) will be invoked. Since it deals with the isColorblind dimension and then removes it before calling drawPixel again, method (1) will end up being called with the color mapped to grayscale, yielding the correct overall result.

However, method (2) could have dealt with and removed the location dimension instead:

```
method {                                                (2a)
  rcvr ≤ screenParent,
  isColorblind ≤ true,
  location ≤ southernHemi
}
drawPixel(x, y, c) {
  {-location}.drawPixel(x, -y, c);
}
```

In this case, the message above would not have worked as intended.This drawPixel method would have been found as the most specific and, since it removes the location dimension, the specialization for Antarctica would not be executed at all. A new, most-specific method would need to be written:

```
method {
        rcvr ≤ screenParent,
        isColorblind ≤ true,
        location ≤ antarctica
}
drawPixel(x, y, c) {
        {-location}.drawPixel(2*x, -2*y, c);
}
```

It is possible that this problem could be solved by using a construct like *super* or *call-next-method* instead of removing the location dimension. Defining such a construct for the symmetric, multidimensional world of Korz, in which dimensions are unordered and considered equal, is challenging, and remains an issue for future research.

The design decision for additional dimensions to trump inheritance was made to facilitate a common and powerful form of evolution in Korz, illustrated earlier in the colored point example: the addition of dimensions. Our experience early on suggested that this precedence allows such evolution to happen more gracefully in many cases, without the need for writing additional, more-specific methods. This rule works especially well when the approach used in this example is followed: handling a new dimension by doing something and then calling the method again without that dimension. The effect of this approach is analogous to *around advice* and *proceed* in AspectJ [Kicz01]. However, our choice may not always do what might be desired.

The design decision also has the interesting effect that a slot that is less specific in one context may not match at all in another. Referring back to the formal definition in Section 3.1.8, for dimension binding set *dbs* and dimension constraint sets *dcs* and *dcs*',

$$dbs \sqsubseteq dcs \wedge dcs \preccurlyeq dcs' \nRightarrow dbs \sqsubseteq dcs'$$

because *dcs*' could have fewer dimensions but tighter constraints on some of the dimensions it does have, tight enough that *dbs* does not satisfy it. This might be confusing to programmers when they take a global view of slot specificity, independent of any specific message send, and then consider a particular message with the global view in mind. The confusion is removed if one focuses on a particular message, considering only slots that match that message.

To avoid these possibilities of confusion, it might be possible and desirable to change the rule so that specificity of shared dimensions is always considered for dimension constraint set specificity, even in the case that extra dimensions are present. We expect that this would introduce many more ambiguity errors, each requiring a more-specific method to be written. This can be annoying, but does help to highlight situations where the programmer's intuition, based on experience with other languages, may not match the semantics of Korz. More research and experience are needed.

Appendix C: More on Multidimensional Context Issues

The incorporation of symmetric, multidimensional, implicit context into Korz raises issues which we explore in more depth here.

C.1 Symmetry and Subjectivity

The issues of symmetry and subjectivity become clearer when one thinks in terms of the multidimensional slot space. Figure 5 shows three dimensions of the slot space for the color point example of Section 4. Each dimension shows the coordinates that are appropriate to it, as well as a special don't-care indicator ("-"). Each slot is positioned in

this space based on the coordinates specified in the dimension binding set of its slot guard. For any dimension not mentioned or not constrained in the slot guard, the don't-care position is used. The figure shows the position of the first slot in Figure 4, whose dimension constraint set is:

{ rcvr ≤ screenParent, location ≤ southernHemi }

Since isColorblind is not included in the constraint set, this slot is shown in the don't-care position for this dimension.

The space is symmetrical in that there is no dominant dimension that determines the program structure. Instead, the developer can have the IDE present a variety of asymmetrical, subjective views of this space that are appropriate for different purposes. For example, for working on screen display issues in general, the view in Figure 6 presenting screenParent as an object with all relevant slots is best; for focussing on support for location, views like Figure 7 presenting southernHemi and other locations as an object is best; and for working on accessibility and ensuring that colorblind users are well supported, a view presenting true as an object is best, ideally filtered to focus on the isColorblind dimension (since true is likely to be widely used as a coordinate). Each of these views is obtained by cutting through the slot space a different way, restricting one's view to a plane (or, in general, a region) that is relevant to one's current task. They can even be combined as in Figure 4, which shows both hierarchies and reifies the slots.

C.2 Thorny issue: 'rcvr' in Korz

Korz's use of the dimension rcvr is at odds with two Korz principles: that a single receiver is replaced by a multi-dimensional context, and that all dimensions are treated equally. Let's use the example in section 4.1 to examine this issue.

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 and ugly enough that we will keep thinking about this issue until we solve it more satisfactorily.

One possible solution would be to define methods like copy as global methods that take an explicit parameter: Instead of

```
method { rcvr } copy() { ... }
```

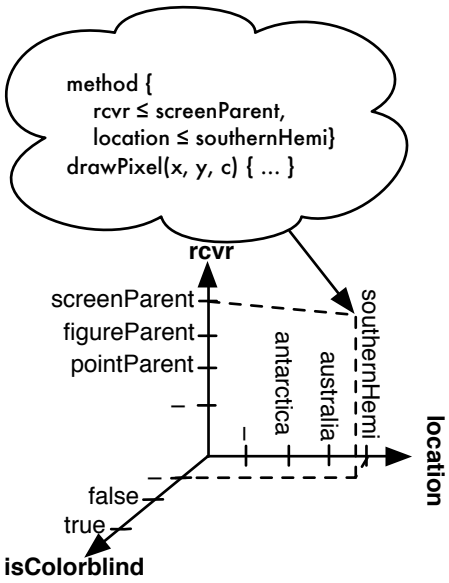

Figure 5: Three dimensions of the slot space for the color point example

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 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:

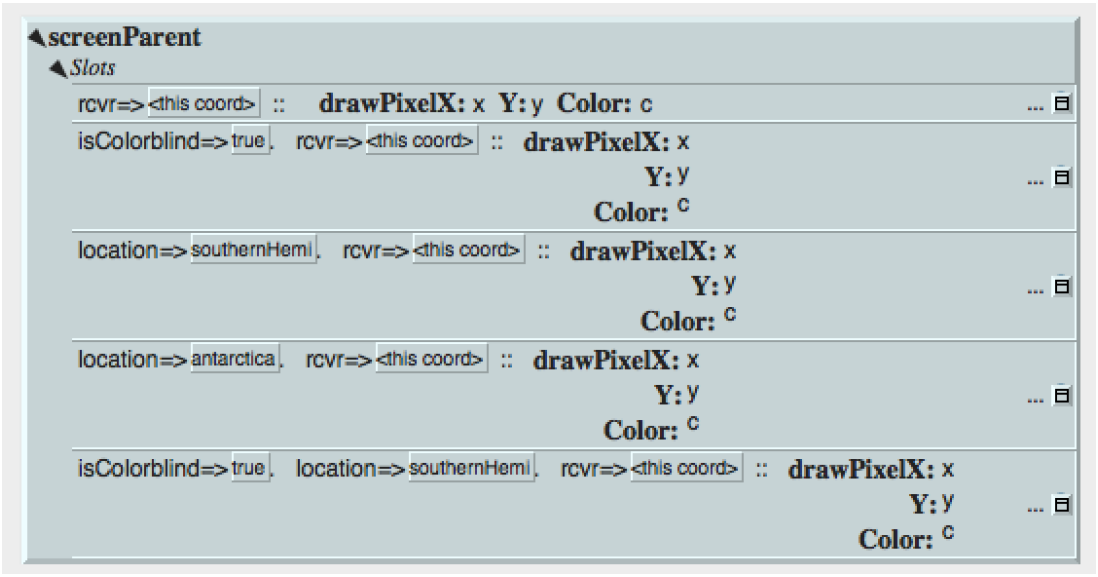

Figure 6: IDE view of screenParent as an object

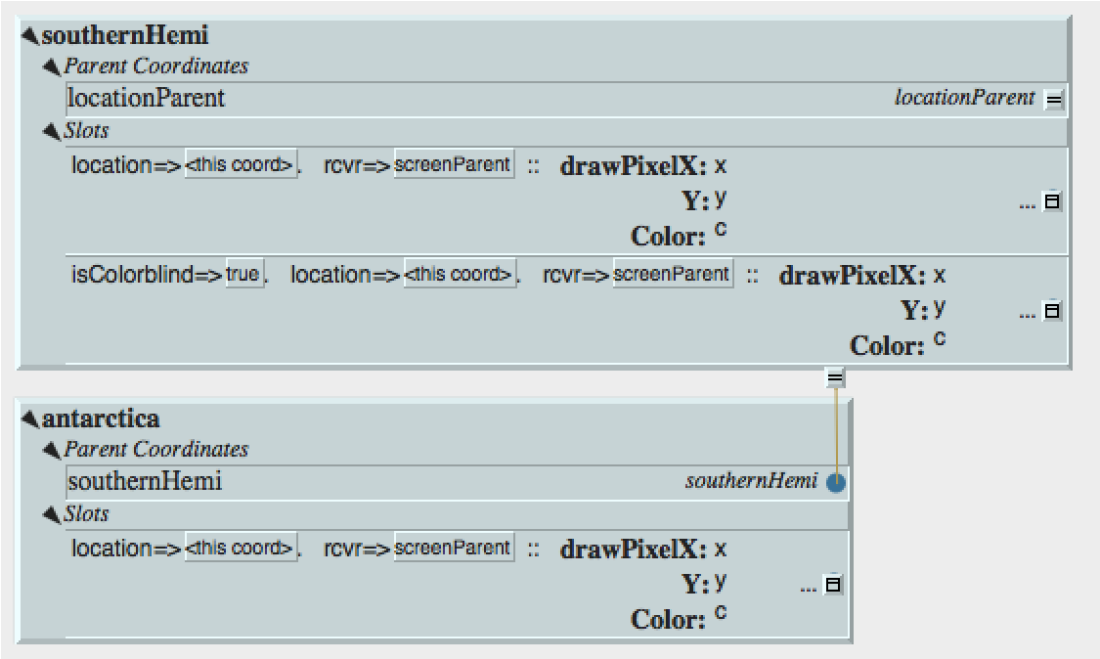

Figure 7: IDE view of locations as objects

```
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. This viewpoint dependence is within the spirit of the Korz IDE supporting subjectivity.

In short, there is much interesting research yet to be done towards providing true symmetry of dimensions and also convenience and familiarity for programmers, especially those with backgrounds in object-oriented languages. For the present, we reluctantly use rcvr, which provides the familiarity and convenience, allows 1-dimensional Korz programs to look exactly like object-oriented programs, and supports convenient addition of "ancillary" dimensions.

Appendix D: Speculating on Modeling and Programming Issues

Korz's expressive power comes at a price: the need for more help from the IDE.

D.1 Slot space versus object model

Traditional object-oriented programmers, when wanting an overall understanding of a program, think in terms of an *object model,* in which the inheritance hierarchy plays a key role in organization and overall understanding. In Korz, the multidimensional slot space assumes this role. To have an overall understanding of a program, a developer needs to understand what the dimensions are, and what coordinates are appropriate on each. Coordinates can have parents, so each dimension effectively has an inheritance hierarchy. This overall view, which is imparted by visualizations such as the one shown in Figure 5, identifies the important concepts in the domain of the program, and indicates what combinations of cases are being considered. To some extent, it serves as an interface. When writing code, one of the key issues is what options are available for use in a message context, the analogy of what operations are supported by an object in an object-oriented language. For example, when coding a `drawPixel(x, y, color)` message send, what options are available for the `rcvr` dimension, and is it sensitive to different choices for `location` or `isColorBlind`? The structure of the space indicates what options are potentially available, and views that show how the space is populated indicate what options are actually available. Such views can be dynamically produced by the IDE.

The space can also be a useful guide to implementers. What cases must be considered and implemented? Suppose, for example, that one is working to ensure proper support for colorblind users across an application. One can then focus on the `isColorBlind` dimension. Any slot whose coordinate in this dimension is `true` already supports color blindness, and any slot whose coordinate is explicitly `false` presumably provides behavior suitable for people who can distinguish colors. The slots in the don't-care position in this dimension are worthy of examination. The developer can look at each and decide whether color blindness is relevant or not, and act accordingly. Analysis performed by the IDE could help, for example to identify selectors that have no methods with `isColorBlind ≤ true` yet are related, according to some metric, to slots that do. The fact that the slot space makes these choices manifest leads to thoughts of such analyses, and can be expected to facilitate their implementation.

Multidimensional spaces are conceptually simple and regular, but quickly become large and hence complex in detail. This leads to concern that they will confuse rather than help programmers. Certainly sophisticated IDE support is critical to working with them effectively. However, it is important to note that they do not introduce complexity; rather, they manifest inherent complexity. A simple program that does not deal with many areas of variation will have a simple space with few dimensions, perhaps even none. As areas of variation arise, as they invariably do in real-life programming (and real life in general), more complex structures and dependencies are inevitable, and often the dependencies are somewhat ad hoc, because only immediately-needed cases are considered. In most programs, these dependencies are hidden in the code and are easy to miss, or worse, hidden in requirements or design documentation and never explicitly referenced in the code. This makes it difficult to amass the knowledge of the program needed for evolution tasks, and makes all but the simplest evolution tasks dangerous, because it is easy to miss something. The multidimensional structure of Korz make more of the inherent structural complexity and dependencies manifest, and encourages regularity (or at least can highlight irregularity). Hence we believe it has the potential to reduce the effort and the risks in evolution tasks. More research is needed to test this belief.

D.2 Modularity

The issue of modularity in Korz is especially interesting. More research is needed, but this section gives some brief, informal thoughts.

On one hand, dimensions provide a flexible and powerful modularization mechanism, that can be used for program organization and presentation, as described above, and also has presence at runtime. 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.

On the other hand, the dimension names in Korz are global. This presents problems if one needs to merge two Korz slot spaces that have some dimension names in common, especially if those names are used with different meanings in the two spaces. IDE support can help here, providing for renaming of dimensions that should be different, and handling mapping of coordinates in dimensions that should be merged. However, it is an open question as to whether this sort of approach is adequate, or whether Korz should provide additional mechanism, such as encapsulation of entire slot spaces, or namespaces for dimension names.

D.3 Static analysis and programmer assistance

The fact that Korz programs consist of large numbers of small pieces (slots) means that the programmer is likely to need help finding things when needed, and avoiding mistakes. At the same time, the dynamic nature of Korz suggests that there are limits to how much help can be provided statically.

The dimensions provide valuable structural information to programmers, and identify immediately key areas of variability. A simple analysis of the slot space can reveal the dimension names, and the sets of coordinates actually used in each dimension at any point in time. The results of this

analysis can be used to provide intelligent code completion in slot guards and message sends.

Though Korz is not statically typed, the constraints in slot guards do provide a good deal of information that can be used for type inferencing. In addition, Agesen demonstrated that it is possible to analyze Self programs so as to provide the programmer with assistance and checking such as is normally expected only in statically-typed languages [AU94], and we believe the approach can be extended to Korz.

These and related issues require further research.

Appendix E: Relationship to SOP, MDSOC and AOP

This section explores in more detail the relationship of Korz to subject-oriented programming (SOP), multidimensional separation of concerns (MDSOC) and aspect-oriented programming (AOP).

Subject-oriented programming (SOP) [HO93] pioneered the notion of subjective objects, and modules called *subjects* were the precursors of COP layers. Any given object had fixed identity across a system. Different subjects could associate slots, both data and methods, with objects. Overall behavior was determined by composing subjects according to programmer-specified *composition rules* [Ossh96]. This composition resulted in instantiation of composed objects at runtime, embodying the combined state and behavior of the composed subjects, but each subject had its own, restricted, subjective view of these objects. The original SOP paper [HO93] described dynamic activation and de-activation of subjects during execution, much like COP layers. However, implementations of SOP all performed subject-composition prior to execution.

SOP was extended with multidimensional structure in multidimensional separation of concerns (MDSOC) [Tarr99]. The paper observed that programs typically suffer from the *tyranny of the dominant decomposition*, where the programming language supports one particular way of modularizing programs, such as by object or class. Concerns that align with the dominant decomposition are well modularized, but other concerns (such as features in object-oriented languages) cannot be; their code ends up being scattered across many modules. SOP, aspect-oriented programming (AOP) [Kicz97] and related approaches [Elra01], and feature-oriented approaches (FOP) [Bato94, Preh97] had improved the situation by introducing a second form of module (subject, aspect, etc.) that allowed additional kinds of concerns to be modularized. MDSOC went beyond two dimensions. Whatever modules were used in a program, e.g., classes, subjects or aspects, slots were considered to be arranged in a multidimensional space, like that of Korz. The space was called a *hyperspace*, and hyperplanes in this space, called *hyperslices*, could be extracted at will, and composed into *hypermodules* by means of composition relationships, analogous to the composition rules of SOP. Hypermodules were themselves composable (in fact, hyperslices were just primitive hypermodules). This broke the tyranny of the dominant decomposition, allowing *on-demand remodularization:* flexible configuration and reconfiguration of software to satisfy new requirements or to support new evolution tasks, irrespective of the original modularization.

MDSOC was implemented in Hyper/J [OT00] and the Concern Manipulation Environment (CME) [Chun05a, Chun05b, Harr05], both of which performed the remodularization and composition before execution. Korz adds dynamicity and conceptual simplicity by embracing the multidimensional space as the runtime program representation, and using multiple dispatch rather than a separate composition step. However, as of now, it does not support the richness of composition operators, such as the ability to to execute multiple methods, each contributed by a different hyperslice, in response to a single message. Another limitation relative to MDSOC, and especially AOP, 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, but does not cover the case of selector-based patterns. In section 8 we briefly discussed how small extensions to the Korz model which, we believe, make sense in general, have the potential to overcome the limitations with respect to method composition/ combination and selector-based patterns.