Predicated Generic Functions
Enabling Context-Dependent Method Dispatch

Jorge Vallejos[1], Sebastián González[2], Pascal Costanza[1], Wolfgang De Meuter[1], Theo D'Hondt[1], and Kim Mens[2]

[1] Software Languages Lab – Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
{jvallejo,pascal.costanza,wdmeuter,tjdhondt}@vub.ac.be
[2] Département d'ingénierie informatique – Université catholique de Louvain
Place Sainte-Barbe 2, 1348 Louvain-la-Neuve, Belgium
{s.gonzalez,kim.mens}@uclouvain.be

Abstract. This paper presents predicated generic functions, a novel programming language abstraction that allows the expression of context-dependent behaviour in a declarative and modular manner, providing fine-grained control of method applicability and method specificity. Methods are guarded by predicates with user-defined orderings, thereby increasing the expressiveness of existing method dispatching approaches. We have prototyped our proposal in Lambic, an extension of the standard Common Lisp Object System. We illustrate and motivate our approach by discussing the implementation of a collaborative graphical editor.

1 Introduction

The lack of linguistic support for encoding context-dependent behaviour forces programmers to scatter these dependencies throughout application code in the form of conditional statements. In object-oriented programming, ad hoc polymorphism alleviates this problem by means of dynamic method dispatch, enabling behavioural variations based on a *receiver* argument. Methods and their overriding relationships are driven by the inheritance hierarchies of the objects received as arguments. Although an improvement, object-oriented dispatch has been found to be limiting in many situations. A number of solutions have been proposed, ranging from design patterns and multiple dispatch to those based on metaobject protocols and aspects. Still, most approaches do not offer linguistic abstractions to influence the semantics of method dispatch based on more general criteria, while still preserving the benefits of encapsulation and polymorphism —with one remarkable exception, *predicate dispatch* [11].

Predicate dispatch offers fine-grained control on method applicability by means of logical predicates. Logical implication between predicates defines the overriding relationship between corresponding methods. However, predicate dispatching has limited capacities to resolve method overriding ambiguities since the logical implications between predicates cannot be decided in the general case. Thus, the set of method predicates must be restricted to a well-chosen subset

and thus can be statically analysed. This leads to a viable approach, but can be limiting in some circumstances, since users cannot extend predicate dispatching with their own arbitrary predicates in a straightforward way. Predicates without logical implications can still be added but they are treated as black boxes and the overriding relationship between two syntactically different expressions is considered ambiguous [5, 11].

This paper proposes a generic generic function-based multiple dispatch mechanism, called *predicated generic functions*, that alleviates the restrictions of predicate dispatching to cope with method overriding ambiguities. Instead of requiring a logical implication order between predicates, this model fosters the definition of context-specific priorities. Predicated generic functions enable users to establish a priority order between logically unrelated predicates. These priorities are specified in a per-generic-function basis: predicated generic functions contain not only the methods with a common name and argument structure (as in standard generic function models [3, 7]), but also the predicates on which such methods can be specialised. A method is selected for execution when its predicate expression is satisfied, and the order of the predicates specified in a generic function determines the order of applicability of its methods.

We implement the mechanism of predicated generic functions in *Lambic* [22], a prototype extension of the Common Lisp [19] programming language. We illustrate the benefits of predicated generic functions by developing a scenario of context-aware computing. Lambic allows application developers to modularise behavioural adaptations in methods and declaratively specify the context conditions for these adaptations as predicates. Manual definition of predicate priorities in generic functions provides developers with fine-grained control over the composition of adaptations, ensuring that the "most suitable" composition of behaviour is selected by the dispatch mechanism for any given method call.

2 Motivation: Context-Dependent Behaviour

Context dependency is the ability of software services to perceive and react to changes in their execution environment, adapting their behaviour accordingly [14]. This ability is already an integral part of some business applications, but it is becoming even more critical in application domains such as mobile and ubiquitous computing, in which context adaptability requirements play a central role.

In this section we show the need for language constructs that ease the expression of context-dependent behaviour, discussing the suitability of existing object-oriented approaches to cope with this requirement. We use as running example a distributed graphical editor called *Geuze*. This editor can be used in different hosts to work collaboratively on a same graphical document. For the sake of simplicity, in this section we focus only on the graphic user interface of Geuze, and discuss other cases of context dependency we have found in the implementation of this editor, in Section 5.

Table 1. Context conditions for Geuze operations

Context	**GUI events**		
	Mouse down	**Mouse move**	**Mouse up**
Painting shape	shape found, brush selected	—	—
Moving shape	shape found, brush not selected	drag status found, brush not selected	drag status found, brush not selected
Drawing shape	shape not found, brush selected	drag status not found	line status found
Drawing selection	—	brush not selected, drag status not found	brush not selected, drag status not found
Selecting shape	shape found	—	—
Deselecting shape	shape not found	—	—

2.1 Handling User Interface Events

Consider some of the main graphical operations that Geuze can perform. The editor allows creating, selecting, moving and painting shapes in a canvas. Each operation has its own interaction pattern defined in handlers for Graphical User Interface (GUI) events. A pattern can require a combination of GUI events, and the same GUI event can be used in several patterns. For instance, the operation for painting a shape is defined in a handler for the *mouse-down* event (clicking inside a shape with the mouse pointer paints the shape). The operations for drawing and moving a shape follow a drag-and-drop pattern, which is specified in handlers for the *mouse-down*, *mouse-move* and *mouse-up* events.

The context of use plays a key role, as it determines the operation that handles an event. This is illustrated in Table 1. For example, the operation that should be executed upon a *mouse-down* event depends on context information such as the x and y coordinates of the mouse pointer, on the *shape* found at these coordinates (if any) and on the state of the editor's *brush* (whether the brush button is selected or not). Depending on this information, the operation corresponding to the *mouse-down* gesture might be painting, moving and so forth.

The mode of operation (*painting*, *moving*, etc.) is also part of the context. Pressing the mouse button will trigger a different set of actions depending on the currently active operation, as illustrated in Table 2. For instance, a same *mouse-move* event provokes a displacement of the shape when the editor is in *moving* mode, whereas a line is drawn —a completely different behaviour— when the mode is *drawing*. Further, there are cases in which two operations correspond to the same *mouse-down* event. If a deselected shape is painted, the shape is first selected, and then painted. This order of operations, although not apparent in Tables 1 and 2, is integral part of the normal behaviour of the editor and needs to be properly encoded. We will come back to this when we discuss the implementation in Lambic.

Table 2. Actions for Geuze operations

Context	Actions		
	Mouse down	**Mouse move**	**Mouse up**
Painting shape	paint shape	—	—
Moving shape	set drag status	update drag status, move shape	delete drag status
Drawing shape	set line status, draw initial point	update line status, draw line	delete line status, create a shape object out of the drawn line
Drawing selection	—	draw selection square, select found shape	remove selection square
Selecting shape	select shape	—	—
Deselecting shape	deselect shape	—	—

2.2 Design Analysis of Running Example

At first sight, programming a basic graphical editor such as the one described previously is straightforward. Nevertheless, a detailed analysis of the possible solutions reveals that the programming tools we have at hand today fall short of expressiveness to allow the production of a cleanly modularised solution.

Naive solution The most immediate implementation of the handler for the *mouse-down* event is through conditional statements. Such a handler would squeeze all the information contained in Tables 1 and 2 into one monolithic piece of code: both the conditions that are necessary for execution of context-specific behaviour, and the behaviour itself, for every operation of the editor that is concerned by the *mouse-down* event. The downsides are clear —context conditions would be hard coded using conditional statements, and the implementations of the different context actions would be tangled in the handler. This solution is unacceptable, as it hinders maintainability and code reuse [15].

Object-oriented solution In traditional object-oriented systems, method invocation is triggered by messages being sent to objects where the objects then decide which method to execute based on a mapping from message signatures to actual methods. Typically, such mappings are fixed for specific types of objects, which means that the dynamic state of a running system cannot (easily) influence the dispatch mechanism for a particular object anymore.

One solution is to use forwarding, which means that an object that receives a message forwards it to another object, based on some arbitrary criteria. A popular example of that approach is the State pattern [12], which enables separation of method definitions according to the state of a particular receiver object. Fig. 1 shows a diagram for a possible use of the State pattern in our example. Using this architecture, the editor may behave differently according to whether it is in the state *painting*, *moving*, and so forth.

A drawback of message forwarding is that it introduces object identity problems: the `self` or `this` reference in the method corresponding to the current state is not the original receiver of the message (note in Fig. 1 that the first argument

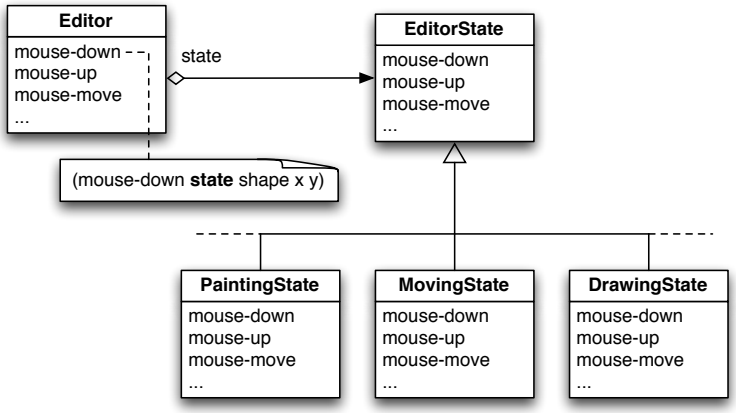

Fig. 1. Architecture of the Geuze editor using the State pattern.

passed to *mouse-down* is the `state` attribute of the editor). This is typically referred to as the object schizophrenia problem [6]. There are a number of suggestions to solve certain aspects of this problem, for example by rebinding `self` or `this` to the original receiver in delegation-based languages [16], or by grouping delegating objects in *split objects* and letting them share a common object identity [1]. However, the core problem remains, namely that it is not straightforward to unambiguously *refer* to a single object anymore. The programmer must ensure that the right object in a delegation chain is being referred to, and even in split objects, the correct *role* of an object has to be selected.

Predicate dispatching Predicate dispatching [11] is very convenient for the declaration of *method applicability* constraints: the conditions —or context— under which a method *can* be invoked. However, it falls short in helping to express *method specificity* in the general case —that is, when methods *should* be invoked, taking precedence over other applicable methods. Precedence by logical implication is a natural choice, but it sometimes cannot be established by mere static analysis of the predicates. Unfortunately, predicate dispatching does not support user-defined orderings of predicates for cases that cannot be decided solely on the grounds of the structure of the predicates. Furthermore, automatic disambiguation of methods by means of logical implication does not always yield the desired semantics. For instance, in Table 1 the condition for *moving shape* is stronger than (implies) that of *selection*. Predicate dispatching will thus consider moving behaviour more specific than selection behaviour. However, in Geuze the selection behaviour must be performed *before* the moving behaviour.[3] Lambic allows to encode this semantics, as will be shown in the explanation of Listing 1.

In this section we have shown that it is difficult to cleanly encode context-dependent behaviour within the traditional object-oriented paradigm. We have

[3]This requirement has to do with the distributed part of the editor, not explained in this paper. The selection of a shape is used for control access. Selecting a shape means obtaining a lock from the leader that coordinates the interaction in the network.

identified predicate dispatching as a possible solution path, but have discussed its shortcomings when it comes to defining method specificity. Next we introduce our answer to still use the underlying idea of predicated applicability, albeit in an adapted form that also allows fine-tuning of specificity.

3 Predicated Generic Functions

Predicated generic functions are an extension of the generic function-based mechanism of the Common Lisp Object System (CLOS) [3]. CLOS supports multiple dispatch semantics by detaching methods from classes, allowing developers to specialise methods on the classes of all received arguments, as opposed to only the first argument in singly dispatched languages. We extend such mechanism by enabling methods to also specialise on predicates. In this section, we briefly explain the syntax and informal semantics of predicated generic functions, showing a small example of use. To this end, we use *Lambic* [22], our extension of the Common Lisp programming language that implements predicated generic functions. In the remainder of this paper we refer to predicated generic functions and to Lambic interchangeably.

3.1 Syntax and Semantics

This section describes the very essentials of Lambic's syntax and semantics for the definition and invocation of predicated generic functions and methods. Further details of this mechanism are available in Section 5.

Defining Generic Functions and Methods In Lambic, as in CLOS, a generic function is a container of methods with a common name and a parameter list the methods can specialise. Additionally, Lambic's generic functions can contain a list of predicate declarations. The definition of such generic functions follows the syntax:

(**defgeneric** *function-name function-parameters*
  [(**:predicates** {*pred-symbol* | (*pred-name pred-params pred-body*)}*)])

A generic function is defined with the `defgeneric` construct which receives as arguments a name, a parameter list and an optional list of predicate declarations (denoted with the `:predicates` keyword). Predicates are standard (CLOS) functions with a boolean-valued expression as body, following an arbitrary user-defined specificity order represented by the order of the predicate declarations in the generic function: the first predicate of the list is the most general predicate and the last one the most specific. A predicate can be defined either outside or inside the generic function. In the former case it suffices to indicate only the symbol associated to the predicate's definition (which can correspond to an already existing function). Internally defined predicates, on the other hand, should indicate the predicate's name, parameter list and body. These internal predicates are available exclusively for the methods belonging to the generic function.

Methods are defined independently from their containing generic functions, using the `defmethod` construct:

(**defmethod** *method-name method-parameters*
  [(**:when** {(*pred-name arguments*)}*)]
  method-body)

A method is defined with a name, a parameter list and an optional predicate expression (specified with the `:when` keyword). This expression is composed of one or more invocations to the predicates declared in the method's generic function, using one or more parameters of the method as arguments.

Invoking Generic Functions In Lambic, as in CLOS, object-oriented programs are written in terms of generic function invocations rather than messages exchanged between objects. Yet, both approaches result in the invocation of a method or a method chain. When a generic function is called with particular arguments, it selects the methods to be executed —known as the *applicable methods*— by evaluating each of the method's predicate expressions. This evaluation occurs in the lexical environment of the method, augmented with the generic function's parameters bound to the received arguments. The applicable methods are those whose predicate expression evaluates to `true`, and the methods that do not specify any predicate.

The execution order of the applicable methods is determined by the specificity of their predicates: the method with the most specific predicate is executed first. A method without predicate expression is considered more general than any method with predicates and thus it is always executed at last.

Finally, the most specific method is called. The other methods can be invoked by the programmer inside of method definitions by way of `call-next-method`, much like with super calls in other object-oriented languages.

3.2 Example of Use

Consider as illustrative example the factorial function. In this function we want to distinguish between negative and positive numbers, and the number zero. We therefore define a `factorial` generic function using as predicates the functions for aritmethic comparison `<`, `=`, and `>`. Since these are already defined in Common Lisp, we just need to declare them as predicates for the `factorial` generic function, indicating the corresponding symbols as follows:

```
(defgeneric factorial (n)
  (:predicates < = >))
```

We can now define methods for this generic function:

```
(defmethod factorial (n)
  (:when (> n 0))
  (* n (factorial (- n 1))))

(defmethod factorial (n)
  (:when (= n 0))
  1)
```

```
(defmethod factorial (n)
  (:when (< n 0))
  (error "Factorial not defined for negative numbers."))
```

Each of these methods indicates one of the predicates declared in the generic function using the `:when` keyword. The first method is called if the argument `n` is a positive number and computes the general case of the factorial function. The second method is called if `n` is 0 and returns 1. The third method will be called if `n` is a negative number and signals an error.

4 The Geuze Editor in Lambic

We now describe the development in Lambic of the scenario introduced in Section 2, the Geuze drawing editor. First, we focus on the implementation of a GUI event handler to illustrate the benefits of the predicated generic functions in terms of modularity and reusability. We then show that by supporting method dispatch based on the state of received arguments, Lambic enables expressing State-like patterns without object identity problems.

4.1 Context-Dependent Event Handlers

Listing 1 shows the implementation in Lambic of the actions that handle the *mouse-down* event in the Geuze drawing editor. Each of these actions correspond to (part of) a graphical operation, which is selected depending on a number of context conditions, detailed in Table 1. In Lambic, we can cleanly separate such actions in methods specialised on predicates representing the different graphical operations. These methods act as *context-dependent* handlers which can conveniently associate actions to context conditions. For instance, the first method definition in Listing 1 describes how to handle the *mouse-down* event when the editor is in the *painting* context (indicated by means of the `painting` predicate).[4]

All the methods in Listing 1 are contained in the `mouse-down` generic function which declares the predicates with the context conditions for the graphical operations. Listing 2 shows the definition of such generic function with the `moving` and `drawing` predicates defined internally. The other three predicates are defined as external functions as in the case of the `selecting` predicate shown in Listing 2. Section 4.2 explains the reason for this difference in the declaration of the predicates.

Combining Context-dependent Handlers In Section 2, we discuss the situation in which an event is handled by more than one operation, e.g. painting a deselected shape results in the shape first being selected and then painted. In Lambic, the selection and proper combination of methods is internally computed in accordance to the predicates and their order of declaration in the generic

[4] For the sake of clarity, the handling action in this case is reduced to the invocation of a `paint-shape` method.

Listing 1. Untangled GUI event handlers in Lambic. Application concerns are indicated on the right side.

```
(defmethod mouse-down (editor shape x y)
  (:when (painting shape editor))                    → context
  (paint-shape editor shape))                        → painting

(defmethod mouse-down (editor shape x y)
  (:when (moving shape editor))                      → context
  (set-drag-status editor x y))                      → moving

(defmethod mouse-down (editor shape x y)
  (:when (drawing shape editor))                     → context
  (set-line-status editor x y)
  (draw-point editor x y))                           → drawing

(defmethod mouse-down (editor shape x y)
  (:when (selecting shape editor))                   → context
  (select-shape editor shape)                        → selection
  (call-next-method))

(defmethod mouse-down (editor shape x y)
  (:when (deselecting shape editor))                 → context
  (deselect-shapes editor))                          → deselection
```

function. Hence, in the Geuze editor, such case is transparently handled by the `mouse-down` generic function, which selects for execution the methods defined for selecting and painting shapes (shown in Listing 1), denoted with the `selecting` and `painting` predicates respectively. Since the `selecting` predicate is declared more specific than the `painting` predicate (the latter predicate appears first in the list of predicates of the `mouse-down` generic function), the method for selecting the shape is executed first. Finally, as we explained in Section 3, by default only the most specific method is executed. Therefore, we need to include the `call-next-method` construct in the method specialised on the `selecting` predicate, so that the one specialised on `painting` is invoked next.

4.2 State Pattern without Object Schizophrenia

Lambic enables developers to express State-like patterns without object identity problems. As example, assume that the graphical operations of the Geuze editor represent its different states, just like in the original State pattern. Each state groups the behaviour required by an operation to handle one or more GUI events, as detailed in Table 2 of Section 2. In Lambic, this corresponds to describing a state as a number of methods using the same predicate expression. Simple examples are the states representing the operations for selecting, deselecting and painting shapes, which only require one method definition to handle the *mouse-down* event, shown in Listing 1. A bigger example is the *moving* state

Listing 2. The `mouse-down` generic function.

```
(defgeneric mouse-down (editor shape x y)
  (:predicates painting
               (moving (shape editor)
                 (and shape (not (brush-active? editor))))
               (drawing (shape editor)
                 (and (not shape) (brush-active? editor)))
               selecting
               deselecting))

(defun selecting (shape)
  shape)
```

Listing 3. The *moving* state.

```
(defmethod mouse-down (editor shape x y)
  (:when (moving shape editor))
  (set-drag-status editor x y))

(defmethod mouse-move (editor shape x y)
  (:when (moving shape editor))
  (move-shape shape editor x y))

(defmethod mouse-up (editor shape)
  (:when (moving shape editor))
  (delete-drag-status editor))
```

presented in Listing 3 which defines its behaviour in the methods `mouse-down` (to set a drag status used during the move), `mouse-move` (to move the shape) and `mouse-up` (to remove the drag status at the end of the move).

Notice how this way of specifying the behaviour of the Geuze editor cleanly separates the definition of its several states (embodied by the predicates) from the behaviour corresponding to those states. This State-like idiom avoids any object identity problems: a particular editor always retains its identity, no matter what state it is in. Since the state of the editor is automatically derived from the current context conditions, one does not have to worry about managing an explicit state with explicit state switches in the corresponding *mouse-down*, *mouse-move* and *mouse-up* event handlers.

Finally, note in Section 2 that Table 1 identifies different context conditions that characterise the operation for moving a shape at the different mouse events. In particular, there is a set of conditions for the *mouse-down* event and another set for the *mouse-move* and *mouse-up* events. However, we can still define the *moving* state in terms of the three methods using the `moving` predicate. In Lambic, this is possible by enabling predicates to be defined inside the generic functions which are only available to the generic function's methods. Thus, the `mouse-move` and `mouse-up` generic function refer to a globally defined `moving` predicate (as shown in Listing 4) whereas the `mouse-down` generic func-

Listing 4. Reuse of the `moving` predicate.

```
(defun moving (shape editor)
  (and (drag-status? editor) (not (brush-selected?))))

(defgeneric mouse-move (editor shape x y)
  (:predicates ... moving ...))

(defgeneric mouse-up (editor shape x y)
  (:predicates ... moving ...))
```

tion defines its own version of such predicate (shown in Listing 2). This example makes clear that the State pattern in Lambic is mostly a naming convention for the predicate used to identify the state. Further, this example shows the fined-grained control that developers have to influence the applicability of methods based on the context.

5 Validation and Discussion

Lambic's predicated generic functions allow methods to specialise on programmer-defined predicates, providing fine-grained control of method applicability, as predicate dispatching also does. Additionally, method dispatch is driven by the predicates' specificity order declared in generic functions which avoids the problems caused by potential ambiguities when comparing arbitrary predicates that do not designate instance subsets of each other.

We have illustrated the benefits of predicated generic functions by discussing the implementation of the graphical user interface of the Geuze editor. However, this is only part of our current implementation of Geuze.[5] As mentioned in Section 2, Geuze is an application for collaborative edition that enables its users to create peer-to-peer drawing sessions. Predicated generic functions have contributed significantly to define context-dependent behaviour for this application, allowing a clean separation of the code required for collaborative work from the plain editor logic, a clear distinction between the behaviour for the roles required for the coordination of the session (the session leader and the rest of the participants), and the modularisation and dynamic composition of the graphical operations (which is discussed in this paper). In the implementation of Geuze we have used 4 instances of the State-like idiom described in Section 4.2 to handle events related to network connection (to deal with network failures), synchronisation (to control edition locks), replication (to ensure a consistent collaborative edition), and graphic user interface (partially shown in Section 4.2). None of these instances require additional infrastructure beyond the methods associated to states by means of context predicates. Geuze is composed of 44 methods grouped in 16 generic functions. None of these methods contain entangled concerns in their body.

[5] The full implementation is available at `http://soft.vub.ac.be/lambic`.

5.1 Predicated Generic Functions and CLOS

In the current implementation of predicated generic functions in Lambic we have adopted a rather conservative approach which preserves the method dispatching semantics of CLOS. For instance, Lambic still enables the methods to specialise on the classes of the arguments, and to use qualifiers (e.g. `:around`, `:before` and `:after`). We achieve this compatibility by reflectively introducing our predicate dispatch mechanism as an internal step in the method selection and ordering process, leaving unchanged the rest of the semantics of CLOS that computes applicable methods and their ordering. Actually, predicate expressions correspond to implicit generic function arguments which are added at the end of the parameter list. Each predicate is associated to a type, and the predicate ordering is encoded by means of subtyping. Given this encoding, behaviour selection is performed following normal CLOS dispatch semantics. In particular, a generic function containing methods specialised on classes and predicates, selects and sorts the methods according to the classes first, and then according to predicates. Our current implementation of Geuze[5] illustrates this case.

5.2 Limitations

Although Lambic can help in tackling some of the challenges for modelling generic functions with context-specific predciates, a number of challenging issues needs to be further explored. Currently, predicated generic functions are implemented using the Meta-Object Protocol (MOP) of CLOS, and can be used only in the LispWorks® Enterprise Edition[6] development environment for Common Lisp. We have not considered efficiency issues in detail yet, and have not explored more general implementation techniques. However, efficient implementation techniques for generalised predicate dispatch have been investigated in detail in the past [5] and can probably be adapted to the implementation of predicated generic functions as well.

In this paper, we propose an alternative to logical implication order used by existing predicate dispatching approaches to disambiguate method overriding. However, there are situations in which such approach would still be desired, e.g. to disambiguate methods using the same predicate expression. For instance, using predicated generic functions the method definitions

```
(defmethod foo (n)
  (:when (> n 1))
  (print “Number bigger than 1”))

(defmethod foo (n)
  (:when (> n 2))
  (print “Number bigger than 2”))
```

would lead to an ambiguous situation if `foo` is invoked with `n` greater than 2, as both methods would be selected for execution but none of them is more specific

[6] See `http://www.lispworks.com`.

than the other. While Lambic avoids this problem by not allowing the definition of methods with the same predicate expression, this is clearly a case in which the inclusion of logical implication in Lambic would increase its expressiveness.

6 Related Work

We divide related techniques in two categories, according to whether they promote flexible method dispatch or flexible software composition to enable behaviour adaptability.

6.1 Flexible Behaviour Selection Schemes

Predicated generic functions build on previous work on filtered dispatch [9], which in turn draws inspiration from specialisation-oriented programming [18]. Filtered dispatch sticks to inheritance for the definition of specialisation relationships between objects, and introduces *filter expressions* that map actual arguments to representatives of equivalence classes. Filtered arguments are then used in place of the original arguments for method selection and combination. The chosen method is invoked using the original arguments.[7] As Lambic's predicates, filters are associated to generic functions.

Filter expressions can function as plain dispatch predicates, by returning `t` (true) to signal applicability, and `nil` (false) to prevent execution of a method, much as the predicates of Lambic do. There are however significant differences in both approaches. Firstly, even though many filters can be defined for a given generic function in filtered dispatch, corresponding methods can use only one of those filters at a time. As a consequence, each possible combination of the filters that could prove useful needs to be anticipated and encoded as an additional filter in the generic function. Secondly, filtered expressions are parametrised exclusively on the argument they filter; they cannot depend on the value of other arguments of the method. This restriction renders filtered dispatch less amenable to express context adaptations, because the conditions for applicability (the predicates) cannot harness all available contextual information —namely, the information contained directly or indirectly in the arguments for which the method invocation is being requested.

Mode classes [20] enable dispatching on the explicit state of an object. Predicate classes [4] extend this idea by dispatching on computed states. Mode classes correspond to an explicit management of state, while predicate classes compute state implicitly. Predicate classes were a precursor to generalised predicate dispatch [11], discussed in Section 2.2.

Clojure[8] is a recent dialect of Lisp that provides a generic dispatch framework that can accommodate many different semantics. A Clojure multimethod

[7] This corresponds to the *lookup* ∘ *apply* decomposition of method dispatch investigated by Malenfant et al. [17]. In filtered dispatch, *lookup* receives the filtered arguments, and *apply* receives the unfiltered (original) ones.

[8] See `http://clojure.org`.

is a combination of a *dispatching function* and one or more *methods* with the same name. The dispatching function maps passed arguments to arbitrary values which are associated to methods. The ordering of values combines Java's typing mechanism with Clojure's own ad-hoc hierarchy system. When no method dominates the others, Clojure provides a means to manually disambiguate multiple matches. An implementation of predicated generic functions in the dispatch framework of Clojure seems feasible.

Ambience [13] proposes a prototype-based object system that reifies the context as an object, and exploits multiple dispatch to enable selection of context-dependent behaviour. Ambience does not propose however a predefined mechanism to choose which context should be activated; in contrast, in Lambic the conditions that enable different behaviours are readily encoded in the method predicates.

6.2 Dynamic Composition Schemes

Classboxes [2] are a mechanism for lexically-scoped structural *refinements*. Similar to open classes, refinements in classboxes make it possible to extend a class definition from the outside, with new fields and methods, as well as extending methods with a mechanism similar to overriding in standard object-oriented programming. However, while changes made with open classes are globally visible, classboxes introduce a dynamic scoping mechanism based on import relations between modules: a refinement to a class is only visible for all execution that originates from a client of the module in which the refinement is defined.

Aspect-Oriented Programming (AOP) [15] allows the programmer to encapsulate concerns that cross-cut modularisation boundaries (e.g. classes) in a construct called an *aspect*. Aspects are designed to supplement the basic composition mechanisms provided by the host language. There are three main binding times for aspects: compile time, load time and run time. Dynamic aspect weaving occurs at run time. Dynamic aspect weaving can be thought of as a tool for dynamic behaviour adaptation, since it allows for base application logic to change at run time —for instance, to be adapted to non-functional concerns such as security or low power computation. Dynamic aspects can be woven and unwoven according to context. Context-Aware Aspects explore this idea [21]. The idea is to extend pointcut languages with context-specific restrictions, allowing both parameterization of context definitions and exposure of context state to the aspect action. This and other aspect weaving approaches are related to ours insofar as the applicability of advice is determined by dynamically evaluated pointcut definitions —advice is analog to filtered methods, and pointcuts are analog to predicates which limit the applicability of advice to specific contexts.

Costanza [8] argues that the advantages of (dynamic) AOP can be obtained with less conceptual and technical burden thanks to dynamic scoping of functions. This idea has been integrated reflectively into CLOS, and the result is ContextL [10]; ContextL features a form of dynamic scoping of methods to enable adaptation to context.

7 Conclusion and Future Work

In this paper we have introduced *predicated generic functions*, a novel mechanism to allow flexible behaviour selection according to context. This mechanism has been realised in Lambic, an extension of CLOS. Lambic method definitions can be guarded by predicates, which are used to decide on the applicability of the method for a list of actual arguments. If more than one predicated method is applicable, the order in which the predicates are declared in the corresponding generic function is used as tiebreaker. These are the main tools Lambic offers for fine-grained control of applicability and specificity of methods.

The development of Geuze, a non-trivial application for the collaborative edition of graphical objects, has shown us that predicated generic functions allow the modular implementation of various concerns. The modes of operation of the application (whether it is working on *painting*, *moving* or some other mode) can be regarded as contexts, and behaviour is specialised on such contexts.

The assessment of predicated generic functions presented in this paper leads us to believe that they are well suited to the expression of behaviour that depends non-trivially on context. However, some work remains. Among our next steps, we plan to look into efficient implementation techniques to avoid unnecessary computation when deciding method applicability. Also, we will consider the formalisation of the semantics of predicated generic functions. Finally, we will continue exploring the possibilities of predicated generic functions in combination with distribution —one of the flagships of Lambic, offering many possibilities yet to be explored.

References

1. Bardou, D., Dony, C.: Split objects: A disciplined use of delegation within objects. ACM SIGPLAN Notices 31(10), 122–137 (1996)
2. Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Classboxes: Controlling visibility of class extensions. Journal of Computer Languages, Systems and Structures 31(3), 107—126 (Dec 2005)
3. Bobrow, D., DeMichiel, L., Gabriel, R., Keene, S., Kiczales, G., Moon, D.: Common Lisp Object System specification. Lisp and Symbolic Computation 1(3/4), 245–394 (1989)
4. Chambers, C.: Predicate classes. In: Proceedings of the European Conference on Object-Oriented Programming. Lecture Notes in Computer Science, vol. 707, pp. 268–296. Springer-Verlag (1993)
5. Chambers, C., Chen, W.: Efficient multiple and predicated dispatching. In: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. pp. 238–255. ACM Press (1999)
6. Chandra Sekharaiah, K., Janaki Ram, D.: Object schizophrenia problem in object role system design. In: Lecture Notes in Computer Science. vol. 2425, pp. 1–8. Springer-Verlag (2002)
7. Clifton, C., Leavens, G., Chambers, C., Millstein, T.: MultiJava: Modular open classes and symmetric multiple dispatch for Java. In: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications. pp. 130–145. ACM Press (2000)

8. Costanza, P.: Dynamically scoped functions as the essence of AOP. ACM SIGPLAN Notices 38(8), 29–36 (2003)
9. Costanza, P., Herzeel, C., Vallejos, J., D'Hondt, T.: Filtered dispatch. In: Proceedings of the Dynamic Languages Symposium. ACM Press (Jul 2008), co-located with ECOOP'08
10. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: an overview of ContextL. In: Proceedings of the Dynamic Languages Symposium. pp. 1–10. ACM Press (Oct 2005), co-located with OOPSLA'05
11. Ernst, M., Kaplan, C., Chambers, C.: Predicate dispatching: A unified theory of dispatch. In: Proceedings of the European Conference on Object-Oriented Programming. Lecture Notes in Computer Science, vol. 1445, pp. 186–211. Springer-Verlag (1998)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing Series, Addison-Wesley (1995)
13. González, S., Mens, K., Heymans, P.: Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In: Proceedings of the Dynamic Languages Symposium. pp. 77–88. ACM Press (Oct 2007), co-located with OOPSLA'07
14. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology 7(3), 125–151 (March–April 2008)
15. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proceedings of the European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, vol. 1241, pp. 220–242. Springer-Verlag (1997)
16. Lieberman, H.: Using prototypical objects to implement shared behavior in object-oriented systems. ACM SIGPLAN Notices 21, 214–223 (1986)
17. Malenfant, J., Dony, C., Cointe, P.: A semantics of introspection in a reflective prototype-based language. In: LISP and Symbolic Computation. vol. 9, pp. 153–179. Springer Netherlands (May 1996)
18. Newton, J., Rhodes, C.: Custom specializers in object-oriented Lisp. Journal of Universal Computer Science 14(20), 3370–3388 (2008)
19. Steele, G.: Common Lisp: The Language. Digital Press, second edition edn. (1990)
20. Taivalsaari, A.: Object-oriented programming with modes. Journal of Object-Oriented Programming 6(3), 25–32 (Jun 1993)
21. Tanter, E., Gybels, K., Denker, M., Bergel, A.: Context-aware aspects. In: Software Composition. vol. 4089, pp. 227–242. Springer-Verlag (2006)
22. Vallejos, J., Costanza, P., Van Cutsem, T., De Meuter, W.: Reconciling Generic Functions with Actors. In: ACM SIGPLAN International Lisp Conference, Cambridge, Massachusetts (2009)