Method Combinations for Programming Languages with Multiple Dispatch

Supervisor / Contact	Pr. Didier Verna <didier.verna@epita.fr>
Company / Lab	EPITA Research Lab
Work Address	14–16 rue Voltaire, le Kremlin-Bicêtre, France
Dates	*Flexible*
Funding	Probable 750€/month gross unless self-funded (to be confirmed)
Keywords	Object-Oriented Programming, Generic Functions Multiple Dispatch, Orthogonality

Context

In the traditional Object-Oriented (OO) approach (Smalltalk, and later C++, Java, *etc.*), methods syntactically belong to classes and the dynamic dispatch mechanism selects the appropriate (most specific) one based on the class of the object through which the method is invoked. For example, in a C++ or Java call such as `object.method(...)`, the concrete class of `object` is retrieved at run-time, and its super-hierarchy is traversed bottom-up until a matching `method` is found. This process known as "inclusion polymorphism", is also called "message-passing" or "single dispatch" [10]: method selection is based on a *single* `object`, the receiver of the message.

Single dispatch is known to have its limitations. In particular, it does not play well with static type safety [3] and is not suited to situations in which there is no single privileged receiver for a message, as in symmetrical operations such as binary methods [2]: should we write `obj1.equal(obj2)`, or `obj2.equal(obj1)`?

Multiple dispatch [4, 5] is a form of dynamic dispatch in which *any* number of arguments can be specialized and used for method selection. If we syntactically write `method(this, ...)` instead of `object.method(...)`, it becomes apparent that multiple dispatch is a more general form of polymorphism: single dispatch is simply multiple dispatch with only the first argument specialized. Because multiple dispatch doesn't grant any argument a particular status (message receiver), methods (herein called "multi-methods") are naturally decoupled from classes and method (or generic function) calls look like ordinary function calls: one now simply writes `method(obj, ...)` or `equal(obj1, obj2)`. The existence of multi-methods thus pushes dynamic dispatch one step further in the direction of Separation of Concerns (SOC) and orthogonality [8]: polymorphism and inheritance are clearly separated from each other.

Common Lisp [11] was the first OO language to be standardized with native support for multiple dispatch through the Common Lisp Object System (CLOS) [6, 9, 1, 7]. Since then, other languages have also adopted multi-methods as a core construct, notably Dylan (a descendant of Lisp) and Julia. It as been shown that multiple dispatch solves many, if not all, of the problems of the traditional OO approach [12, 13].

Yet another improvement over the classical OO approach lies in the concept of "method combination". In the traditional approach, the dynamic dispatch algorithm is hardwired: every polymorphic call ends up executing the most specific method available (applicable) and using other (less specific) ones requires calling them explicitly, which is sub-optimal. Indeed, as soon as a method needs to call another one, as in `Class::method(...);` for C++, `super.method(...);` for Java, or even `(call-next-method)` for Lisp, the method in question ends up doing two different things at the same time: its actual job, and a form of ad-hoc manual dispatch. Method combinations make it possible for generic functions to use custom dispatch algorithms, hereby removing the need for explicit method chaining. For example, a generic function using the `and` method combination would implicitly call *all* the applicable methods (not necessarily by order of specificity), and combine their results with a logical `and`.

Method combinations were originally introduced in CLOS. At the time, the concept was not completely stabilized, but it has been recently refined [14, 15]. Along with multiple dispatch, method combinations constitute one more step towards SOC: a generic function can now be seen as a 2D concept: 1. a set of methods and 2. a specific way of combining them. Unfortunately, few languages support them, and even fewer at their core.

Project

The purpose of this internship is to study the applicability of Lisp's method combinations to other programming languages already equipped with multiple dispatch. The applicant will start by producing a comprehensive state-of-the-art review of multiple dispatch support per language (core feature, emulation via a user-level library, *etc.*). The review should also include descriptions of any discovered functionality even remotely connected to the concept of method combinations. For example, some languages have an "advice" mechanism, some have method "modifiers" (`before`, `after`, `around`) which are directly inspired from CLOS's standard method combination qualifiers, *etc.*

In a second step, the applicant will pick one or two languages and implement a proper support for CLOS's standard and built-in method combinations. We are particularly interested in two languages, Python and Julia, but for different reasons. Both of these are quite popular. Multiple dispatch is at the core of Julia's design, so method combinations would be a valuable addition to it. Python's support for multiple dispatch is only available via third-party libraries, but this "limitation" could in fact make it easier to add method combinations. Ultimately however, the choice will be discussed when appropriate.

Finally, note that this project, if successful, will constitute the foundation for an upcoming Ph.D. thesis on the very same topic. In particular, further avenues to explore (including, but not limited to) are core support for method combinations when only emulations exist, support for custom method combinations (as opposed to built-in ones only), investigating method combinations for single dispatch, and the implications of static *vs.* dynamic type checking on the concept.

Profile

The applicant should already have a solid experience in multiple programming languages and OO (Lisp would be a plus, but it is not strictly required), and a deep interest in programming paradigms, SOC, orthogonality, and expressivity in general. The applicant should be already familiar with fundamental concepts of programming language design and implementation such as environments, lexical / dynamic scope, binding, *etc.*. Finally, the applicant should not be afraid of delving into large code bases, either third-party libraries or language sources.

References

[1] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp object system specification. *ACM SIGPLAN Notices*, 23(SI):1–142, 1988.

[2] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. On binary methods. 1(3):221–242, 1995.

[3] Giuseppe Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[4] Giuseppe Castagna. *Object-Oriented Programming, A Unified Foundation*. Progress in Theoretical Computer Science. Birkhäuser Boston, 2012.

[5] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *SIGPLAN Lisp Pointers*, 5(1):182–192, January 1992.

[6] Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An overview. pages 151–170, 1987.

[7] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. CLOS: integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):29–38, 1991.

[8] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[9] Sonja E. Keene. *Object-Oriented Programming in Common Lisp: a Programmer's Guide to CLOS*. Addison-Wesley, 1989.

[10] Kristen Nygaard and Ole-Johan Dahl. The development of the SIMULA languages. *SIGPLAN Notices*, 13(8):245–272, August 1978.

[11] Ansi. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.

[12] Didier Verna. Binary methods programming: the CLOS perspective. *Journal of Universal Computer Science*, 14(20):3389–3411, 2008.

[13] Didier Verna. Revisiting the visitor: the Just Do It pattern. *Journal of Universal Computer Science*, 16(2):246–271, 2010.

[14] Didier Verna. Method combinators. In *11th European Lisp Symposium*, pages 32–41, Marbella, Spain, April 2018.

[15] Didier Verna. A MOP-based implementation for method combinations. In *ELS 2023, the 16th European Lisp Symposium*, pages –, Amsterdam, Netherlands, April 2023.