Mμl: The Power of Dynamic Multi-Methods

(Work-In-Progress Paper)

Isaac Oscar Gariano
Victoria University of Wellington
Wellington, New Zealand
Isaac@ecs.vuw.ac.nz

Marco Servetto
Victoria University of Wellington
Wellington, New Zealand
Marco.Servetto@ecs.vuw.ac.nz

Abstract

Multi-methods are a straightforward extension of traditional (single) dynamic dispatch, which is the core of most object oriented languages. With multi-methods, a method call will select an appropriate implementation based on the values of *multiple* arguments, and not just the first/receiver. Language support for both single and multiple dispatch is typically designed to be used in conjunction with other object oriented features, in particular classes and inheritance. But are these extra features really necessary?

Mμl is a dynamic language designed to be as simple as possible but still supporting flexible abstraction and polymorphism. Mμl provides only two forms of abstraction: (object) identities and (multi) methods. In Mμl method calls are dispatched based on the identity of arguments, as well as what other methods are defined on them. In order to keep Mμl's design simple, when multiple method definitions are applicable, the most *recently defined* one is chosen, not the most *specific* (as is conventional with dynamic dispatch).

In this paper we show how by defining methods at runtime, we obtain much of the power of classes and meta object protocols, in particular the ability to dynamically modify the state and behaviour of 'classes' of objects.

CCS Concepts • **Software and its engineering** → **Object oriented languages**; **Procedures, functions and subroutines**; • **Theory of computation** → *Object oriented constructs*;

Keywords object oriented languages, multi-methods, dynamic dispatch, meta-object-protocols

ACM Reference Format:
Isaac Oscar Gariano and Marco Servetto. 2019. Mμl: The Power of Dynamic Multi-Methods: (Work-In-Progress Paper). Presented at *Workshop on Meta-Programming Techniques and Reflection (META'19).* 6 pages.

META'19, October 20, 2019, Athens, Greece

1 Introduction

Most object oriented (and functional[1]) languages support only *single* dynamic dispatch: the code executed by a method call is determined by a *single* argument (the receiver). For example, to evaluate a call like x`.add(`y`)`, the value of x will be inspected for a definition of an `add` method (depending on the language, this may be found in a slot/field [1], class [8], and/or parent object of x [16]) . This approach can be inflexible for two main reasons: the value of y is ignored when selecting the implementation of `add` and only the creator of x can define an `add` method.

Multi-methods [2, 4, 6, 10, 11, 13–15, 17] are one approach to overcoming these limitations: methods can be declared to dispatch based on the values of *multiple* arguments. Unlike conventional single dispatch, multi-methods can usually be declared outside of the objects/classes of any of their arguments. Consider for example the following Julia code:

```
add(x :: Number, y :: Number) = x + y
add(x :: Array, y :: Array) =
    [add(xy[1], xy[2]) for xy = zip(x, y)]

add(x :: Array, y :: Number) = [add(xe, y) for xe = x]
add(x :: Number, y :: Array) = [add(x, ye) for ye = y]

add([[1, 2], 3], 4) // evaluates to [[5, 6], 7]
```

Thus a call like `add(`x`, `y`)` will first get the types of x *and* y, say T_1 and T_2, and then execute the body of the `add` method defined with `(_ ::` T_1`, _ ::` T_2`)` (or `(_ ::` T_1'`, ..., _ ::` T_2'`)`, for some super types T_1' and T_2' of T_1 and T_2, respectively).

In the above example, `Array` and `Number` are core language types, yet we were allowed to define the various `add` methods; this is in contrast to conventional single dispatch languages, in which users cannot easily write methods that dynamically dispatch over pre-defined types/classes.

Mμl is a language built around multi-methods as an attempt at being even more flexible by eliminating as many extra concepts as possible, whilst increasing the flexibility of multi-methods.

[1] Since a function or lambda can be thought of as an object with a single 'apply' or 'call' method.

1.1 The Mμl Programming Language

The language of Mμl[2] is designed to be as simple as possible, it is an expression based left-to-right call-by-value language with the following grammar (where x and μ are identifiers and ι is an 'identity'):

$$\begin{array}{rrl} \text{(Expression)} & e & ::= t \mid \texttt{new} \mid \texttt{def}\ M \mid \mu(\overline{e}) \\ & & \quad e; e' \mid x := e; e' \\ \text{(Term)} & t & ::= x \mid \iota \\ \text{(Method)} & M & ::= \mu(\overline{p} \mid \overline{c})\ \{e\} \\ \text{(Paramater)} & p & ::= t \mid {=}t \\ \text{(Constraint)} & c & ::= \mu(\overline{t}) \end{array}$$

A *(ground) term* (t) is either a standard variable name (x) or an *identity* (ι); during reduction, an x may be replaced by an ι. *Identities* are the only kind of value in Mμl, their only intrinsic meaning is whether they are the same or different (i.e., their 'identity'). An ι cannot appear in the source code, and can only be created by a `new` expression, which will evaluate to a fresh identity that is distinct from any pre-existing ones.; thus identities cannot be forged.

Methods (M) are global and are dynamically installed by an expression of the form $\texttt{def}\ \mu(\overline{p} \mid \overline{c})\ \{e\}$, or simply $\texttt{def}\ \mu(\overline{p})\ \{e\}$ when the $\overline{c}$ is empty. Here μ is the name of the method being defined, however *multiple* methods may be defined with the same name; the $\overline{p}$ and $\overline{c}$ are (comma separated) lists of parameters and constraints (respectively); and the body of the method is given by e. The e, $\overline{p}$, and $\overline{c}$ are interpreted in the scope where the `def` was evaluated, augmented with appropriate bindings for any p_i of form x: i.e. methods are closures, as with traditional lambda expressions. A `def` expression does not reduce to a value/identity, and so cannot be used as an *argument* to a method call or the RHS of a ':=', however it can be used as the *body* of another method[3].

A *method call* expression is of the form $\mu(\overline{e})$ and will first evaluate its arguments to identities $\overline{\iota}$ and then reduce to $e'[\overline{p := \iota}]$, where $\mu(\overline{p} \mid \overline{c})\ \{e'\}$ is the *most recently* installed applicable method, and for each p_i of form x, $[\overline{p := \iota}]$ performs standard capture-avoiding substitution of ι for x. A method $\mu(\overline{p} \mid \overline{c})\ \{e'\}$ is *applicable* to the call $\mu(\overline{\iota})$ whenever each $\overline{p}[\overline{p := \iota}]$ *accepts* the corresponding $\overline{\iota}$, and each $\overline{c}\ [\overline{p := \iota}]$ is *satisfied*. When no such applicable method exists, reduction will get stuck.

A *parameter* (p) of form t will accept any value, whereas one of form $={t}$ will only accept *the current* value of t. A *constraint* (c) is of form $\mu(\overline{t})$ and is satisfied whenever it has a matching applicable method: i.e. whenever the call $\mu(\overline{t})$ is defined.

Finally, Mμl supports standard *sequence* expressions of the form $e; e'$, and let expressions of the form $x := e; e'$. We present a formalised set of reduction-rules in Appendix A.

The following example demonstrates the aforementioned features in action:

```
def foo(n) { n };
// create three distinct identies
bar := new; baz := new; qux := new;
foo(bar); foo(baz); foo(qux); // evaluates to bar, baz, and qux

def foo(=bar) { baz };
foo(bar); // Now this evaluates to baz
foo(qux); // Still evaluates to qux, as qux ≠ bar

def foo(x | foo2(x)) { foo2(x) };
foo(bar); // Still evaluates to baz, as foo2(bar) is undefined

def foo2(=bar) { bar };
foo(bar); // now returns foo2(bar), which returns bar
```

Later on we will introduce syntax sugar for natural numbers, lists, and lambdas. Our examples also use string literals of the form `"text"` which evaluate to an identity ι such that `print(`ι`)` will print *text*.

We have implemented Mμl[4], with all the aforementioned features, in Racket [5] by using the 'brag' [18] parser generator and an encoding of our reduction rules in PLT Redex [9]. We have used this to test that all the code examples presented in this paper behave as indicated.

2 Abstraction Support

We now show how Mμl is powerful enough to encode various language features including mutable fields, 'value equality' (as opposed to reference equality), multiple dynamic dispatch, and multiple inheritance. Similarly to meta-object-protocols, Mμl allows such features to be used at runtime, allowing state and behaviour to be dynamically added to whole groups of objects at a time, thus providing support for meta-programming. None of this power requires any extensions to Mμl, the two core abstractions of identities and methods are sufficient; this is in contrast to most languages where such power is provided by specific abstractions and syntactic forms.

2.1 (Mutable) State

Suppose we want to represent records: collections of named (mutable) values; we can do this by representing the fields as methods (which are closures):

```
def new-point(x, y) { // a 'constructor' for a 'record'
    res := new;
    def get-x(=res) { x }; // a 'getter'
    def get-y(=res) { y };
    // res is like a {x = x, y = y} record
    res };

point := new-point(1, 2);
```

[2] 'Mμl' is pronounced like 'mull'; the μ stands for the ancient Greek word μέθοδος ('methodos'), meaning pursuit of knowledge.

[3] This restriction is to keep the formalism simple, and we leave it future work to provide a meaningful value for such an expression.

[4] The source code is available at github.com/IsaacOscar/Mul.

```
get-x(point); // returns 1, like point.x
```

The above works because in Mμl method definitions, unlike variables and parameters, have global scope. We can update fields by re-defining methods (compare this with languages where fields are also methods [1]):

```
def get-x(=point) { 2 }; // like point.x = 2
get-x(point); // now returns 2
```

Or, better yet, we can define a setter:

```
// can only call this method if get-x(that) is already defined
def set-x(that, x | get-x(that)) {
    def get-x(=that) { x } };

set-x(point, 3);
get-x(point); // now returns 3
```

This works because in Mμl it is not an error to redefine a method: a method call will resolve to the *most recently* defined applicable method (in this case, the one defined by the most recently installed `get-x(=point)` method). We could have instead placed a `def set-f1(=res, x)` inside the body of `point`, however the `set-f1(x, y | f1(x))` version gives us code reuse: if an identity ι has a corresponding `get-x(`ι`)` method, it will automatically get a `set-f1(`ι, y`)` one, even if ι is not the result of a call to `new-foo`.

Note that we can also provide a `global-set-x(`x`)` method that will change the value of `get-x` for *all* identities:

```
def global-set-x(x) {
    def get-x(that) { x } };

global-set-x(4);
get-x(point); // now returns 4
```

This is similar to changing an instance slot to a shared slot using the meta object protocol of Common Lisp [2].

2.2 Value Equality

In the previous section we used integer literals, we can obviously simply represent these as method calls to some 'successor' method, but how do we ensure that a `2` appearing somewhere in the source code is the same identity as another `2`? Of course we can let the compiler 'intern' these, but there's a more general way, memoise the successor function:

```
zero := new; // An identity simply representing the number 0

def succ(n) { // returns an identity representing n + 1
    res := new;
    def succ(=n) { res }; // memoise the result
    def pred(=res) { n };
    res };
```

Thanks to the `def succ(=n){res}` line above, any future calls to `succ`, with an identical `n` value will return an identical result. Thus we can safely desugar a number literal n into `succ`^{n}`(zero)`.

We can also easily define arithmetic operations such as `plus` in terms of `zero`, `succ`, and `pred`:

```
def plus(x, =zero) { x }; // Since x + 0 = x
def plus(x, y | pred(y)) {
    // Since x + (y + 1) = (x + y) + 1
    // This requires pred(y) to be defined (so y ≠ 0)
    succ(plus(x, pred(y))) };

one := succ(zero);
def plus(x, =one) { // optimised case for x + 1
    succ(x) };
```

Note that the same approach can be used to encode lists with value equality:

```
empty := new; // The empty list
def cons(h, t) { // like h : t in Haskell
    res := new;
    def cons(=h, =t) { res }; // memoise the result
    def head(=res) { h }; def tail(=res) { t };
    res };

list1 := cons(1, empty); // Or [1]
list2 := cons(1, empty); // Identical to list1
```

Thus we can desugar lists of the form $[e_1, \ldots, e_n]$ into `cons(`e_1`,...cons(`e_n`,empty)...)`.

2.3 (Multiple) Dynamic Dispatch

One of the simplest tools for dynamic dispatch is the lambda expression, for example consider a classical map function in Haskell:

```
map(f, []) = []; -- case for empty list
map(f, h : t) = f(h) : map(f, t); -- case for non empty list
map(\x -> x + 1, [1, 2]); -- evaluates to [2, 3]
```

We can encode such dynamic dispatch in Mμl by using an `apply` method:

```
def map(f, =empty) { empty };
def map(f, l | head(l)) {
    // Here l ≠ empty, since head(empty) is undefined
    cons(apply(f, head(l)), map(f, tail(l))) };

lam := new; // an identity to represent our lambda expression
def apply(=lam, x) { plus(x, 1) }; // the body of the 'lambda'
map(lam, [1, 2]); // pass the 'lambda'
```

We can use this pattern to desugar lambda expressions of the form $\{\overline{p}$ `=>` $e\}$ to 'l `:= new; def apply(=`$l, \overline{p}$`){`e`};` l', for some fresh identifier l.

Recall our `Number`/`Array` example of multiple dispatch from Section 1; in order to encode this we need to be able to distinguish between a 'natural number' and an 'array/list'. We can do this by using methods:

```
def as-natural(=zero) { zero };
def as-natural(n | pred(n)) { n }; // n = succ(…)
// as-natural(x) is defined iff x is a 'natural number'

def as-list(=empty) { empty };
def as-list(a | head(a), tail(a)) { a }; // a = cons(…)
// as-list(x) is defined iff x is a 'list'
```

Note how our `as-natural`/`as-list` methods behave like 'structural type cast' operations: they return their argument if and only if they have the 'structure' of a 'number'/'array';

here a 'no applicable method' error would correspond to a failed cast[5]. Since we can constrain method definitions by whether such calls would fail, we can use them to encode multiple dispatch:

```
def add(x, y | as-natural(x), as-natural(y)) {
    plus(x, y) };
def add(x, y | as-list(x), as-list(y)) {
    // Assuming zip-map is defined analogously to map
    zip-map({xe, ye => add(xe, ye)}, x, y) };

def add(x, y | as-list(x), as-natural(y)) {
    map({xe => add(xe, y)}, x) };
def add(x, y | as-natural(x), as-list(y)) {
    map({ye => add(x, ye)}, y) };

add([[1, 2], 3], 4); // evaluates to [[5, 6], 7]
```

Thus we still get the power of multiple dispatch without needing a seperate notion of 'type' or 'class': methods themselves can play such a role.

Recall that method definitions apply at *runtime*, and as such, new methods can be dynamically installed; for example if a programmer decides at some point that they wish to provide new overloads for add they can do so:

```
def make-add(x, y, res) {
    def add(=x, =y) { res } };

add(1, 1); // returns 2
make-add(1, 1, 3);
add(1, 1); // now returns 3
```

2.4 (Dynamic) Inheritance

One common use case for classes is (multiple) inheritance: a derived class can automatically get all the methods of its base classes. This is particularly useful as it can significantly reduce code duplication. We can achieve a similar effect by using a technique like that of mixins [3]:

```
def mammal-mixin(self) { // like a 'mammal' class/mixin
    def as-mammal(=self) { self };
    def tails(=self) { 1 };
    def legs(=self) { 4 };
    self };

def biped-mixin(self) {
    def as-biped(=self) { self };
    def legs(=self) { 2 };
    self };

def kangaroo-mixin(self) {
    mammal-mixin(self); // inherit tails and legs from 'mammal'
    biped-mixin(self); // override with 'biped's legs method
    // Note: biped-mixin(mammal-mixin(self)) would also work
    def as-kangaroo(=self) { self };
    def arms(=self) { 2 }; // extra method
    self };

k := kangaroo-mixin(new); // make a new 'kangaroo'
tails(k); legs(k); arms(k); // returns 1, 2, 2
```

We can dynamically perform inheritance by passing an existing object to such a 'mixin':

```
mb := mammal-mixin(new);
legs(mb); // returns 4
biped-mixin(mb); // Almost like adding a new 'class' to mb
legs(mb); // now returns 2
```

We can also use our 'as' methods to do something akin to monkey-patching: we can dynamically install or update a method for all 'instances' of a 'mixin':

```
m := mammal-mixin(new);
legs(m); legs(mb); legs(k); // returns 4, 2, and 2
def legs(self | as-biped(self)) { 3 };
legs(m); legs(mb); legs(k); // returns 4, 3, and 3
```

3 Related Work

Many languages support multi-methods [11], including Common Lisp [2], Clojure [6], Dylan [14], Cecil [4], Julia [13], JavaGI [17], JPred [10], and Korz [15].

Such languages allow methods to specify the required class [2, 4, 6, 14, 17], type [13], parent object [6, 15], and/or a predicate written in a special purpose sublanguage [10]. Unlike Mμl however, these languages come with complicated dispatch algorithms in order to determine the 'most specific' applicable method to call. Notably, Common Lisp, Clojure, and Julia also provide support for runtime redefinition of multi-methods and Common Lisp provides an `(eql e)` specifier which is equivalent to Mμl's $=x$ (since e will be evaluated when the method is defined); however we have not yet found a language that supports something like our $\mu(\overline{x})$ constraints.

Though most of these languages are large with many features, the language of Korz has a similar complexity to Mμl: Korz provides 'coordinates' (like Mμl's identities, except that they can be created with a parent coordinate) and method definitions. It also supports 'dimensions' (similar to implicit parameters in Scala [12]), the values of these 'dimensions' are implicitly passed throughout all method calls, their values can be overridden for individual method calls, and methods can dispatch with respect to them.

4 Future Work & Conclusion

Though the language we presented is both simple and flexible enough to encode many patterns in object oriented languages, there are however several notable limitations in Mμl's design. Firstly, the syntax can become verbose, especially with the common patterns of `def` $\mu(x|m(x))$ and res `:=` `new;` $\mu(=res, \ldots)$ …; res; we are considering various syntax sugars that could alleviate this.

Secondly, Mμl's expressivity could be improved, such as by adding higher-order constraints of the form $m(\mu(x))$ and

[5]This approach has the same problems as structural types: if `head(`*turtle*`)` and `tail(`*turtle*`)` are defined, then so will `as-list(`*turtle*`)`, similarly one can also define an `as-list(=`*frog*`)` method.

providing support for reflective operations, like dynamically querying defined methods. Another major limitation is in our dispatch algorithm: a new method definition will make any pre-existing (less applicable) methods un-callable. This means that one has to be careful as to what order they define methods; in addition, one cannot perform something like a 'resend' or 'next-method' call, nor can we temporarily activate methods, as supported by 'layer activation' in context-oriented programming [7], thus we are unable to simulate 'super' calls.

Finally, and most importantly, Mμl is *too* flexible: we can arbitrarily call define and redefine methods. Not only could this cause programmers to accidentally re-define methods being used by others, it prevents Mμl from fully supporting encapsulation. Though Mμl can prevent arbitrary code from calling 'private' methods, it can't prevent them from being redefined:

```
def person() {
    res := new; inner := new;
    def message(=inner) { // a 'private' method
        "Hello World" };
    def speak(=res) { // a 'public' method
        print(message(inner)) };
    res };

p := person();
speak(p); // Ok, prints "Hello World"
message(p); // Error
def message(x) { "Goodbye World" };
speak(p); // Now prints "Goodbye World"
```

Putting method names in protected 'packages' or allowing handles to specific methods (as in Common Lisp) could help, as could 'final' methods.

Since the easy redefinition of methods makes it very hard to statically determine what a method call will reduce to, it is likely to be seriously difficult to implement Mμl efficiently. We would also like to extend Mμl with something akin to a type-system, so that we can statically ensure the absence of 'no applicable method' errors.

A Reduction Rules

In Section 1.1 we informally described the semantics of Mμl, here we provide a formal definition. For reference, here is the grammar of Mμl together with a standard left-to-right evaluation context $\mathcal{E}$.

$$
\begin{array}{rrl}
\text{(Expression)} & e & ::= t \mid \mathsf{new} \mid \mathsf{def}\ M \mid \mu(\overline{e}) \\
 & & \quad e;\, e' \mid x := e;\, e' \\
\text{(Term)} & t & ::= x \mid \iota \\
\text{(Method)} & M & ::= \mu(\overline{p} \mid \overline{c})\ \{e\} \\
\text{(Paramater)} & p & ::= t \mid {=}t \\
\text{(Constraint)} & c & ::= \mu(\overline{\iota}) \\
\text{(Context)} & \mathcal{E} & ::= \square \mid \mu(\overline{\iota}, \mathcal{E}, \overline{e}) \mid \mathcal{E};\, e \mid x := \mathcal{E};\, e
\end{array}
$$

$$\boxed{\overline{M}|e \longrightarrow \overline{M'}|e'}$$

$$\frac{\overline{M}|e \longrightarrow \overline{M}|e'}{\overline{M}|\mathcal{E}[e] \longrightarrow \overline{M}|\mathcal{E}[e']} \qquad \frac{}{\overline{M}|\iota;\, e \longrightarrow \overline{M}|e}$$

$$\frac{}{\overline{M}|x := \iota;\, e \longrightarrow \overline{M}|e\,[x := \iota]}$$

$$\frac{\iota \text{ is fresh}}{\overline{M}|\mathsf{new} \longrightarrow \overline{M}|\iota} \qquad \frac{}{\overline{M}|\mathsf{def}\ M';\, e \longrightarrow M', \overline{M}|e}$$

$$\frac{\begin{array}{c}\overline{M} \vdash M_i \sim \mu(\overline{\iota}) = e \\ \forall j < i \bullet \nexists e' \bullet \overline{M} \vdash M_j \sim \mu(\overline{\iota}) \longrightarrow e'\end{array}}{\overline{M}|\mu(\overline{\iota}) \longrightarrow \overline{M}|e}$$

$$\boxed{\overline{M} \vdash M' \sim \mu(\overline{\iota}) \longrightarrow e} \qquad \frac{\overline{p[\overline{p} := \overline{\iota}] \sim \iota} \quad \overline{\overline{M} \vdash c[\overline{p} := \overline{\iota}]}}{\overline{M} \vdash \mu(\overline{p} \mid \overline{c})\ \{e\} \sim \mu(\overline{\iota}) \longrightarrow e[\overline{p} := \overline{\iota}]}$$

$$\boxed{p \sim \iota} \qquad \frac{}{\iota \sim \iota}$$

$$\boxed{\overline{M} \vdash c} \qquad \frac{\exists e \bullet \overline{M} \vdash M_i \sim \mu(\overline{\iota}) \longrightarrow e}{\overline{M} \vdash \mu(\overline{\iota})} \qquad \frac{}{{=}\iota \sim \iota}$$

Figure 1. Reduction Rules

We will use the notation $e\,[p := \iota]$ to perform a substitution, where $e\,[{=}t := \iota] = e$ and $e\,[x := \iota]$ performs the usual capture-avoiding substitution; similarly for $p'\,[p := \iota]$ and $c\,[p := \iota]$.

Our reduction rules are presented in figure 1; our arrow is of the form $\overline{M}|e \longrightarrow \overline{M'}|e'$, where the *state* of the system, $\overline{M}$, is the list of installed methods (ordered by most-recent first) and e is the main expression. We use $\overline{M} \vdash M' \sim \mu(\overline{\iota}) \longrightarrow e$ to mean that under state $\overline{M}$, the method M' is applicable to the call $\mu(\overline{\iota})$ and (after substituting in the arguments $\overline{\iota}$) has body e; we use $p \sim \iota$ to mean that the parameter p accepts the value ι; and $\overline{M} \vdash c$ to mean that under state $\overline{M}$, the constraint c is satisfied.

Note that our reduction rules make no attempt at checking for errors, in particular reduction will get stuck if a `def` or an unbound x is used when an ι is expected (such as the RHS of a ':=', an argument to a method-call, or the RHS of an '=' parameter).

References

[1] Martín Abadi and Luca Cardelli. 1995. An imperative object calculus. In *TAPSOFT '95: Theory and Practice of Software Development*, Peter D.

Mosses, Mogens Nielsen, and Michael I. Schwartzbach (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 469–485.

[2] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. 1988. Common Lisp Object System Specification. *SIGPLAN Not.* 23, SI (Sept. 1988), 1–142. https://doi.org/10.1145/885631.885632

[3] Gilad Bracha and William Cook. 1990. Mixin-based Inheritance. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP '90)*. ACM, New York, NY, USA, 303–311. https://doi.org/10.1145/97945.97982

[4] Craig Chambers. 1992. Object-oriented multi-methods in Cecil. In *ECOOP '92 European Conference on Object-Oriented Programming*, Ole Lehrmann Madsen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 33–56.

[5] Matthew Flatt and PLT. 2010. *Reference: Racket.* Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.

[6] Rich Hickey. 2019. Clojure. clojure.org.

[7] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-oriented Programming. *Journal of Object Technology* 7, 3 (March 2008), 125–151. https://doi.org/10.5381/jot.2008.7.3.a4

[8] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (May 2001), 396–450. https://doi.org/10.1145/503502.503505

[9] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. 2012. Run Your Research: On the Effectiveness of Lightweight Mechanization. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 285–296. https://doi.org/10.1145/2103656.2103691

[10] Todd Millstein. 2004. Practical Predicate Dispatch. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*. ACM, New York, NY, USA, 345–364. https://doi.org/10.1145/1028976.1029006

[11] Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. 2008. Multiple Dispatch in Practice. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA '08)*. ACM, New York, NY, USA, 563–582. https://doi.org/10.1145/1449764.1449808

[12] Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2017. Simplicitly: Foundations and Applications of Implicit Function Types. *Proc. ACM Program. Lang.* 2, POPL, Article 42 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158130

[13] The Julia Project. 2019. *The Julia Language.*

[14] Andrew Shalit, David Moon, and Orca Starbuck. 1996. *The Dylan Reference Manual: The Definitive Guide to the New Object-oriented Dynamic Language.* Addison-Wesley Developers Press.

[15] David Ungar, Harold Ossher, and Doug Kimelman. 2014. Korz: Simple, Symmetric, Subjective, Context-Oriented Programming. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 113–131. https://doi.org/10.1145/2661136.2661147

[16] David Ungar and Randall B. Smith. 1987. Self: The Power of Simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA '87)*. ACM, New York, NY, USA, 227–242. https://doi.org/10.1145/38765.38828

[17] Stefan Wehr, Ralf Lämmel, and Peter Thiemann. 2007. JavaGI: Generalized Interfaces for Java. In *ECOOP 2007 – Object-Oriented Programming*, Erik Ernst (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 347–372.

[18] Danny Yoo and Matthew Butterick. 2019. brag: a better Racket AST generato. docs.racket-lang.org/brag.