From APIs to Languages: Generalising Method Names

Michael Homer
Victoria University of Wellington
New Zealand
mwh@ecs.vuw.ac.nz

Timothy Jones
Victoria University of Wellington
New Zealand
tim@ecs.vuw.ac.nz

James Noble
Victoria University of Wellington
New Zealand
kjx@ecs.vuw.ac.nz

Abstract

Method names with multiple separate parts are a feature of many dynamic languages derived from Smalltalk. Generalising the syntax of method names to allow parts to be repeated, optional, or alternatives, means a single definition can respond to a whole family of method requests. We show how generalising method names can support flexible APIs for domain-specific languages, complex initialisation tasks, and control structures defined in libraries. We describe how we have extended Grace to support generalised method names, and prove that such an extension can be integrated into a gradually-typed language while preserving type soundness.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Procedures, functions, and subroutines

Keywords Multi-part method names, application programming interfaces, domain-specific languages

1. Introduction

Multi-part method names, where a method name is a sequence of multiple words each with their own parameters, go back as far as methods like "`between:and:`" and "`to:by:`" in Smalltalk-76 [25]. In this paper we present a generalisation of multi-part names by allowing parts to be repeated, optional, or alternatives, so that a single method defines a whole family of related names. Generalising method names enables advanced APIs and domain-specific languages, blurring the distinction between them. Generalising names also gives concise definitions for families of methods for control structures, such as "`if then`", "`if then else`", "`if then elseif then`". By making explicit the structure of these families of names, generalised methods allow automated error detection both dynamically and statically.

DLS '15, October 25-30, 2015, Pittsburgh, PA, USA.

ACM 978-1-4503-3690-1/15/10. . . $15.00.
http://dx.doi.org/10.1145/2816707.2816708

The next section gives brief background on the Grace language and the motivations for this work. Section 3 describes the design of generalised method names. Section 4 shows and evaluates the application of generalised names within our target use cases, and Section 5 describes the extension of an existing Grace implementation to add them. Section 6 presents a type-safe modification of the TinyGrace [27] formalism to incorporate generalised names into the type system, while Sections 7, 8, and 9 position this paper among related work, suggest future directions, and conclude.

2. Background

Grace is an object-oriented, block-structured, gradually- and structurally-typed language intended to be used for education [4]. Grace includes a number of features that enable the definition of extended or restricted language variants for domain-specific languages or particular teaching paradigms.

Method names in Grace can have multiple parts, as in Smalltalk, but with a more conventional syntax. A method can be declared:

```
method test(val) between(low) and(high) {
  return (val > low) && (val < high)
}
```

This method name has three parts. Each part has its own parameter list, which may contain many parameters, any or all of which may be given types. Each parameter list may also contain a single variadic parameter indicated by a prefix "`*`". The method above could be *requested* with:

```
test(5) between(3) and(10)
```

Multi-part method names allow control structures to be implemented as methods. Like Smalltalk, Grace has no syntactically-privileged control structures: `if then else` is simply a three-part method accepting a boolean and two lambda blocks (written between braces; these are used for all code that may be executed a variable number of times). A dialect [20] can define new control structures to be made available to user code, which will have exactly the same syntax as the built-in structures. Dialects aim to support the design of restricted teaching sublanguages as well as flexible embedded internal domain-specific languages.

3. Generalised Method Names

Generalised method names extend multi-part names so that parts of names and arguments can be optional, repeated or omitted entirely. A generalised name defines a whole (potentially infinite) family of method names within a single definition.

3.1 Use Cases

Several control structures included with Grace have a number of variants. The pattern-matching system of Grace [23], for example, uses multi-part methods to define its `match case...` statement. Many versions of this method are defined, varying only in the number of `case`s. Similarly, many instances of `elseif` and `try catch` are defined, and the same for other structures with many slight variants. These tedious repetitions motivate this work, which will permit a single definition standing in for a whole family of methods.

We wish the system to cover all of the cases above and similar user-defined control structures. In particular, we aim to support these specific uses:

- `match` followed by many `case` parts.
- `try` followed by any number of `catch`es, and then optionally a `finally`.
- `if then` followed by any number of `elseif then` pairs, optionally followed by an `else`.

As a secondary aim, we wish to enable defining more "natural" syntax for complex operations, supporting definitions with user-facing syntax along the lines of Microsoft's LINQ [31, 37], and to expose this ability to domain-specific language authors. We wish a programmer to be able to define a complex fluid interface where parts may be varied, repeated, or omitted according to appropriate rules, and to build an implementation with a minimum of repetition. In all cases, we aim to present meaningful error messages to end users who make mistakes in the construction of their requests, both statically and dynamically.

Finally, we aim to allow authors of libraries with complex initialisation or configuration requirements to build user-friendly, comprehensible, and encapsulated interfaces for these tasks, without resorting to the Builder pattern [17] or complicated method sequences that must be followed.

3.2 Declaration Syntax

We extend the syntax of method declarations to allow indicating that a part may be provided more than once or not at all. No modification is required to the user-level, call-side syntax of the language. We borrow conventional syntax from regular expressions to denote these parts.

A part can be made optional, meaning that a request of this method need not provide that part, using `?`:

```
method a(x) ?b(y) { ... }
```

Table 1. Syntax of generalised method declarations. Any of the prefix operators can be applied to a parenthesised group.

`+b(x)`	One or more `b` parts
`*b(x)`	Zero or more `b` parts
`?b(x)`	Optionally a `b` part
`(b(x) | c(y))`	Either a `b` or a `c` part
`(b(x) c(y))`	A `b` part followed by a `c` part
`?(b(x) c(y))`	Optionally two parts "`b c`" in sequence

This method allows both the requests `a(1)` and `a(1)b(2)`.

A part can be allowed to occur more than once using `+`:

```
method a(x) +b(y) { ... }
```

The above allows requests `a(1) b(2)`, `a(1) b(2) b(3)`, `a(1) b(2) b(3) b(4)`, and so on. Similarly, a `*` indicates that the part may be provided zero or more times.

We permit grouping parts together using parentheses, and then applying any of the operators above to the entire group:

```
method a(x) +(b(y) c(z)) { ... }
```

The above permits `a(1) b(2) c(3)`, `a(1) b(2) c(3) b(4) c(5)`, and so on.

Finally, a choice between particular options (alternation) is permitted using `|` within a parenthesised group.

```
method a(w) (b(x) | c(y) | d(z)) { ... }
```

The above admits requests for all of `a(1) b(2)`, `a(1) c(2)`, and `a(1) d(2)`.

Collectively we refer to all of these as *variable parts*. A method name with no variable parts is *linear*. Variable parts can be nested inside one another, but we require that the body of a variable part begin with a least one ordinary literal part before any further variable parts appear.

3.2.1 Prefixes

We use method prefixes to identify methods, and to match requests to declarations. The *prefix* of a multi-part method name is the longest initial sequence of ordinary parts, plus the prefix of the body of any `+` part that immediately follows. The prefix is thus the sequence of initial request parts that is required in order for a declaration to match a request. Two methods with the same prefix may not be defined in the same object; subclasses' declarations override inherited methods with the same or longer prefix. A method request is mapped to the declaration with the longest prefix. As the prefix of a linear method is the entire method name, all pre-existing methods work exactly as before.

3.3 Requesting Generalised Methods

Requesting a generalised method is indistinguishable from a traditional linear method at the request site, both for requests with an explicit receiver (using the traditional dot no-

tation) and for receiverless requests that are resolved in lexical scope. A request simply includes a list of part names and parameter lists, with no differentiation between variable parts and other parts.

The receiver of a request must determine which method to execute and map the parts of the request to the corresponding parts of the method declaration. A method is identified uniquely by the prefix of its name.

Once the method to execute has been identified, the parts of the request must be matched with the parts of the declaration. We elect to use a greedy approach to this matching: for each part of the declaration, we determine whether it (or the prefix of its body) matches at the start of the remaining unmatched tail of the request, and if so attempt to match the entire part, paying attention to the semantics of each kind of variable part and consuming parts from the request. Within a parenthesised group, we perform the same matching recursively, and within an alternation attempt matching from left to right. If a part is found in the request that we are unable to match, we report an error.

After parts have been matched, arguments are in turn matched to formal parameters. Because the parameters included in variable parts are provided an unknown number of times they are treated in the same fashion as variadic parameters, bound as a sequence. Nested variable parts result in nested sequences.

3.4 Example

This example program illustrates the binding of parameters. This method has one part, `addRatio`, which occurs at least once, and one entirely optional part `multiplyBy`. Its prefix is `use addRatio`.

```
method use(x) +addRatio(y1, y2)
              ?multiplyBy(z) {
  var value := x
  for (y1) and (y2) do { a, b ->
    value := value + a / b
  }
  for (z) do { v -> return value * v }
  return value
}

use(0) addRatio(1, 2) addRatio(2, 4)
       multiplyBy(6) // -> 6
use(1) addRatio(2, 3)
       multiplyBy(2) // -> 3 1/3
```

The literal part `use` always occurs exactly once, and so its formal parameter `x` is bound to the actual argument. As `addRatio` may be provided more than once, both of its formal parameters `y1` and `y2` are bound to sequences inside the method body.

The optional part `multiplyBy` similarly occurs an undetermined number of times, so its parameter `z` is also in a sequence. This sequence will contain at most one item; we elect not to introduce an optional type to represent this case, as it is not required elsewhere in the language.

Variable parts may be nested inside one another, and this sequence-binding occurs recursively. An optional part inside a repeated part, for example, results in its parameters bound to a sequence of sequences: the outer sequence for the repetition, and the inner sequence for the optionality. In this way it is always possible to identify which parts were provided and in what order.

3.5 Greedy Matching

We choose a greedy approach to method-part matching, rather than traditional regular-expression backtracking, because it permits providing useful error messages to the user in the case where the request does not match: we can report concretely "expected to see part X, but instead saw Y" as soon as we meet the issue, rather than reporting a generic match failure later on. We also did not find any use case for more complicated resolution rules; while claiming that method names are defined by a regular grammar makes for a nice paper, practical applications are limited. We prefer to produce a better experience for the end user of a dialect, who may be a novice or non-programmer, rather than allowing (slightly) more freedom to the dialect implementer who understands what they are doing.

This approach means that it is possible to declare a method that can never be requested successfully. For example, the following method is uncallable:

```
method a(x) *b(y) b(z) { ... }
```

The prefix of that method is simply `a`. Once the method is identified, any sequence of `b` parts will be consumed and the provided arguments stored in `y`. Because `*b` has consumed all `b`s greedily, there will be no `b` part in the request left to be matched against the final `b` in the declaration. An error will be raised indicating that the end of the method name was found while a required part was still expected.

Again, we do not consider this to be a real concern. Our use cases all involve the use of different part names, which resolves the issue, and we consider it only a theoretical concern that would indicate particularly poor design in practice; "don't do that, then" remains as solid advice as ever, particularly to the advanced programmers who are the audience for this feature. Regardless, the declaration `a(x) +b(y)` in fact accepts the requests that were presumably the intention of the above.

Our implementation detects some cases of such names that might reasonably occur accidentally and reports static errors at the declaration site, but makes no particular effort to trap all cases. It is similarly possible to declare parts that can never be filled as part of methods that are callable; we do not consider these cases to be errors at all as they may indicate work-in-progress code.

4. Evaluation of Generalised Names in Use

With the extension design in mind we can return to our use cases from Section 3.1. Our motivation had three major cases we wished to support: variadic control structures, complex domain-specific languages, and libraries with large numbers of interdependent configuration options.

4.1 Control Structures

Where previously there were nearly 2,000 lines of definitions of the `match case...` methods (supporting up to 30 `case` blocks), we present here the complete implementation of all variants after the addition of generalised method names, a mere 9 lines:

```
method match(target) +case(cases) {
    for(cases) do { case ->
        def mr = case.match(target)
        if (mr) then {
            return mr.result
        }
    }
    fail "Did not match any case"
}
```

This new version supports an arbitrary number of cases, and is much simpler to understand and modify than what it replaced. Similarly, we are able to define all `elseif then` and `try catch finally` variants simultaneously:

```
method if(bool) then(blk)
        +(elseif(conds) then(blks))
        ?else(elseblk) { ... }
method try(blk) *catch(catchblk)
        ?finally(finallyblk) { ... }
```

These implementations are again much simpler than the complex nesting and repetition that was otherwise required.

4.2 Query Language

Our broadest use case was complex domain-specific languages, and we present a case study of such a language. This language is directly modelled on Microsoft's LINQ for Objects and its syntactic embedding into C♯ and Visual Basic.

We can define a querying method supporting a fluent syntax as in Figure 1. We capitalise the names of parts as "`where`" is a keyword in Grace and is not available as part of a method name. This method admits requests such as:

```
from(students)
    Where { s -> s.enrolledIn "COMP231" }
    Where { s -> s.age > 25 }
    OrderBy { s -> s.gpa }
    Select { s -> s.name }
```

Many filtering and ordering clauses may be provided, and the query may end with either a projection onto the output or a grouping operation. As in C♯ and Visual Basic, this query

```
method from<T>(source : Iterable<T>)
        *Where(filter : Predicate<T>)
        ?(
          OrderBy(ordering : Comparator<T>)
          *ThenBy(tiebreak : Comparator<T>)
        )
        (
          Select(selectProjection
            : Function<T, Any>)
          |
          GroupBy(grouping
            : Function<T, Any>)
        )
        { ... }
```

Figure 1. Generalised query method header.

is simpler to read and write than the equivalent hand-written code, but with generalised method names does not involve a special-case sublanguage in the parser. The implementation is still able to provide meaningful error messages when the syntactic rules of the miniature DSL are broken, as for example by omitting the selection or grouping component, or providing clauses out of order.

The body of the method is reasonably simple given the complexity of the task at hand. Parts that appear at most once are handled trivially:

```
selectProjection.do { p ->
    return data.select(p)
}
```

The more obtuse case is the `ThenBy` clause: because it is within an optional group and may appear many times itself, there are two layers between the parameter name and the actual data:

```
tiebreak.do { ifPresent ->
    for(ifPresent) do { t ->
        data := data.thenBy(t)
    }
}
```

Alternative implementations of this behaviour are possible using the structural knowledge provided elsewhere in the method name. The complete implementation of this query language is included in the samples provided in Section 5.

4.2.1 Discussion

The lack of binding forms manifests here: while macro-based systems (including LINQ itself) are able to introduce new identifiers and rewrite the user's code accordingly, Grace intentionally does not expose such functionality, so that the execution semantics of all code is the same across the board. As a result, the example query includes several blocks `{ s -> ... }` used to thread data items through the

process. These are in some respects unfortunate, but essential to allowing general-purpose extension without rendering code unfollowable. A further language extension permitting the structured introduction of new identifiers as method arguments could ameliorate this issue, but the design of such an extension is out of the scope of this paper.

4.3 Library Initialisation

Our final use case was libraries with complex initialisation requirements, such as widget toolkits. It is common for a given widget to have a great many possible configurations, where only a selection of the options are provided in any given instance but some options form a group that must always be provided together. The interfaces to these configurations fall into three broad categories: the Builder pattern, where an intermediate object supports providing configuration options one at a time before being finalised into the widget object; method sequences, where the object is first created and then configured according to a defined protocol before it may be used; and keyword arguments, where most parameters have default values and can be set by name at construction time. All of these approaches have drawbacks that can be avoided using generalised method names.

All of the configuration options with default values can be made optional parts, with the part names serving to label the role of each argument. Options that must be provided together can be grouped, and complex configuration — placement and initialisation of subwidgets, for example — can be integrated as a user-friendly sublanguage within the main method body, incorporating their own repetitions, alternations or optional parts as required. With these rules in place it will not be possible to omit a contextually mandatory component or perform post-construction configuration operations out of order, while consistent error messages are always presented to the user.

We present the structure of a single common widget from GTK+, a standard button. There are five constructors with slightly different default configurations, and many other configuration options that will commonly be provided in post-constructor setup. The following condenses the constructors and several common post-constructor configuration items into a single method:

```
method button(handler : Block)
    ( ( label(text : String)
      | mnemonic(mnemonicText : String) )
      ?(image(textImage : Image)
        ?position(imagePos:PositionType))
    | namedIcon(name : String,
                size : IconSize)
    | image(image : Image)
    | widget(w : Widget) )
    ?relief(style : ReliefStyle)
    ?alignment(xalign : Number,
               yalign : Number) { ... }
```

We omit further options to save the reader's patience. The above method header is complex, but from the end user's perspective they simply describe what they want:

```
gtk.button { document.save }
    label "Save"
        image(icons.floppy_disk)
        position(gtk.position.right)
    relief(gtk.relief.none)
```

In the standard API, creating this button requires several function calls, with those relating to icon positioning having a required ordering. With the generalised method name, there is a single essentially declarative method request describing the desired outcome: a button that saves when clicked, labelled "Save", with a floppy disk image positioned to the right of the text and no border. It is not possible to try to specify the position of an icon that does not exist, or to try to combine incompatible configurations. We include the full implementation of this method in the distribution.

4.3.1 Discussion

The canonical GTK+ API relies on method sequences to perform this type of configuration, but the Builder pattern could equally be used. A builder can provide a similar level of enforcement to generalised methods, using chained methods instead of a single one, but creating the family of builders to do so is immensely complicated. More commonly, a single builder simply encapsulates the methods of a method sequence, leading to no real improvement over the equivalent sequence API.

A further approach uses keyword arguments with default values. In this case, keyword arguments would be a feasible mechanism, but they do not provide the guarantees about mutual presence or exclusivity that we desire: for example, an image position could be specified regardless of whether an applicable image exists. Such an error will pass silently unless the implementer explicitly tests for the case and raises a run-time error. There is no static or gradual checking of such properties.

Keyword arguments also do poorly when there are multiplicities of arguments involved. In such cases, either list arguments — potentially involving multiple lists which are subsequently paired up, leading to likely misalignment — or a combination of keyword arguments and method sequences is necessary. The generalised method name inherently allows controlled repetition of arguments and syntactic restrictions on what may or must be provided together that pure keyword arguments do not.

Boolean flags are not well supported by any technique, but can be simulated in both the generalised methods and Builder approaches through parts/methods that are given no arguments. In our generalised-methods system these will require empty parentheses `()` after the part name.

4.4 Testing DSL

While LINQ sits on the boundary of library and domain-specific language, we also wish to present a pure DSL study. This language is based heavily on the Gherkin language used by the popular Cucumber [8] testing framework. While Gherkin is "plain text", as the authors found their earlier experience of writing test case stories in Ruby code to be awkward, we will instead embed almost the same natural-language syntax directly into Grace code.

A Gherkin script consists of some number of "features", inside which may be one or more "scenarios" describing individual test cases. A scenario has a name, zero or more "Given:" (setup) clauses, the second and subsequent of which begin with "And:", an event clause "When:", and one or more "Then:" postconditions, the second and subsequent of which again begin with "And:". This grammar fits nicely into our generalised method names, and we can declare a single method to cover an entire scenario and all of its possibilities:

```
method scenario(desc)
        ?(Given(context) *And(contexts))
        When(event)
        Then(expr) ShouldBe(val)
        *(And(exprs) ShouldBe(vals)) {...}
```

We could instead define a method that handled an entire feature, with some number of scenarios included:

```
method feature(name)
        ?background(initialisation)
        +(scenario(desc)
           ?(Given(context) *And(contexts))
           When(event)
           Then(expr) ShouldBe(val)
           *(And(exprs) ShouldBe(vals)))
```

In this case we elect not to do so, because defining a separate `feature` method allows the introduction of new variables in scope of a suite of tests, which we could not do otherwise. Our final language permits defining tests in a form like the following:

```
feature "Addition" do {
    var x
    var y := 1
    scenario "1 + 2 = 3"
        Given { x := 1 }
        And   { y := 2 }
        When  { x := x + y }
        Then  { x } ShouldBe { 3 }
    scenario "1 + 2 = 4"
        Given { x := 1 }
        When  { x := x + 2 }
        Then  { x } ShouldBe { 4 }
        And   { y } ShouldBe { 2 }
}
```

Table 2. Performance of benchmark programs on the unextended baseline and generalised implementations.

	Baseline	**Generalised**	**Slowdown**
Printer	1.744s	1.872s	7.3%
String	1.624s	1.532s	-5.7%

These test cases have a natural-language flow to them, while admitting arbitrary numbers of cases of setup and post-conditions, or omitting the setup entirely, obtained from a single generalised method definition. A complete implementation is included in the distribution from Section 5.

5. Implementation

We extended an existing implementation of Grace, an unoptimising reference interpreter, to add support for generalised method names. This implementation supports all of the variable parts and behaviours described in Section 3. Where the name of a requested method is found to be ill-formed, a constructive error message is reported indicating the part or parts that were expected, and what was found instead.

A distribution of our extended implementation is available from `http://ecs.vuw.ac.nz/~mwh/dls2015.zip`. This distribution includes the studies described in Section 4 and executes on all platforms.

5.1 Performance

The matching of request parts and parameters to generalised method parts and parameters is more complex than for literal method names, and consequently more expensive at run time. To quantify this cost, we performed an experiment running an existing piece of software that made heavy use of "`match` `+case`" pattern matching, and executed it on both modified and unmodified versions of the implementation.

We used a pretty-printing tool that reads Grace source code and outputs it in executable form with semicolons inserted wherever permitted, and gave it its own source code as input. The program was the same on both implementations, and is 519 lines of code. Every node in the parse tree is examined in a 29-case `match case` statement, and several smaller instances occur for particular nodes. The larger statement executes 2,176 times. On the unextended implementation, there were 30 literal definitions of this method, one for each number of cases, while on the extended implementation there was a single generalised definition.

We executed the program 50 times on each implementation and report the mean of all runs. All executions occurred using Mono 4.0.1 on Arch Linux, with glibc 2.21 and Linux 4.0.5 on an otherwise idle machine with an Intel Core i7-4790 CPU at 3.6GHz, using a single core. The results are shown in Table 2.

The execution of this program, a likely worst-case scenario for the extension, took 7.3% longer to complete with generalised methods. Code that does not make heavy use of very long requests of generalised methods in a tight loop

would experience a smaller slowdown, and code that makes no generalised requests at all will not be impacted. In fact, such programs are liable to execute slightly faster, as there are fewer confounding method definitions in existence: a second sample program reversing strings completed 5.7% faster in the generalised implementation. This implementation is entirely unoptimised, and we believe that substantial reduction of this slowdown is possible without compromising the increase in speed given to programs that do not use the generalised methods.

6. Semantics

Grace has optional types, so it is important that generalised methods do not affect the soundness of the type system. The language has a structural subtype system, meaning that one type is a subtype of another if the subtype responds to all the method requests that the supertype accepts with a compatible type. Generalised method names extend the range of methods that can meet requests, and we here formalise this as an extension to Tinygrace [27], an existing formalisation of a subset of Grace, and prove that the resulting type system remains sound.

The grammar of the signature part extension to the standard Tinygrace syntax is given in Figure 2. Note that at least the prefix of any signature (and each of any variants) must begin with a literal part or group. The + modifier has been excluded, as it is sugar for a literal part followed by a *, and would serve only to complicate the formal rules.

We write P as a method part sequence which begins with at least one literal method part or alternation, and V as a single, potentially variable P.

6.1 Subtyping

We overload the subtyping operator $<:$ for all directed compatibility relations. Signature part compatibility is defined in Figure 3. Compatibility between signature parts indicates that any request to a method defined with the parts in the super-sequence will also succeed as a request to a method with the parts in the sub-sequence, which we refer to as *request compatibility*. For instance, we would expect

Syntax

τ ::= **type**$\{\overline{S}\} \mid \ldots$ *(Structural type)*

S ::= $m(\overline{x : \tau})\ \overline{V} \to \tau$ *(Signature)*

P ::= $m(\overline{x : \tau})\ \overline{V} \mid (P \mid P)\ \overline{V}$ *(Signature parts)*

V ::= $^{\mu}P$ *(Variable parts)*

$^{\mu}$ ::= $\epsilon \mid$ * $\mid$? *(Part modifiers)*

e ::= $e.\overline{m(\overline{e})} \mid$ **object**$\{\overline{\textbf{method}\ S\ \{e\}}\} \mid \ldots$ *(Expressions)*

Figure 2. Partial grammar

$\boxed{S <: S}$

(S-SIG)
$$\frac{P_1 <: P_2 \qquad \tau_1 <: \tau_2}{P_1 \to \tau_1 <: P_2 \to \tau_2}$$

$\boxed{\overline{V} <: \overline{V}}$

(S-ONE)
$$\frac{\tau_2 <: \tau_1 \qquad \overline{V_1} <: \overline{V_2}}{m(\overline{x : \tau_1})\ \overline{V_1} <: m(\overline{y : \tau_2})\ \overline{V_2}}$$

(S-SKIP)
$$\frac{P_1 \not<: P_2 \qquad \overline{V_1} <: \overline{V_2}}{[?*]P_1\ \overline{V_1} <: P_2\ \overline{V_2}}$$

(S-LONE)
$$\frac{P_1 <: P_2 \qquad \overline{V_1} <: \overline{V_2}}{?P_1\ \overline{V_1} <: [\epsilon?]P_2\ \overline{V_2}}$$

(S-MANY)
$$\frac{P_1 <: P_2 \qquad *P_1\ \overline{V_1} <: \overline{V_2}}{*P_1\ \overline{V_1} <: [\epsilon?*]P_2\ \overline{V_2}}$$

(S-ALT/LEFT)
$$\frac{P_1\ \overline{V_1} <: \overline{V_2}}{(P_1 \mid P_2)\ \overline{V_1} <: \overline{V_2}}$$

(S-ALT/RIGHT)
$$\frac{P_1 \not<: \overline{V_2} \qquad P_2\ \overline{V_1} <: \overline{V_2}}{(P_1 \mid P_2)\ \overline{V_1} <: \overline{V_2}}$$

(S-ALT)
$$\frac{\overline{V_1} <: P_1\ \overline{V_2} \qquad \overline{V_1} <: P_2\ \overline{V_2}}{\overline{V_1} <: (P_1 \mid P_2)\ \overline{V_2}}$$

(S-EMPTY)
$$\frac{}{\varnothing <: \varnothing}$$

(S-END)
$$\frac{\overline{V} <: \varnothing}{[?*]P\ \overline{V} <: \varnothing}$$

Figure 3. Signature compatibility

a() *(b() | c()) <: a() *b() *c() to hold, because a method which consists of a sequence of any mixture of b()s and c()s can always handle requests to a method which strictly permits a sequence of b() and then c(). The same would not be true in reverse, as the request a() c() b() can be handled by the alternation, but would not be accepted by the other signature.

The compatibility test traverses a part sequence through the modifiers and alternations, eventually matching literal method parts or skipping a variable part in the sub-sequence when the two part names are not the same. Requiring the names to be different is important in the presence of the greedy matching. For instance, it ensures that the sequence ?a() a() is not considered compatible with a(). The greedy matching means that ?a() a() is exactly equivalent to a() a(), which is not compatible with the sequence a().

Parameter compatibility is only addressed by Rule S-ONE, as it is the only rule in the compatibility relation in which two linear method parts are actually matched with one another. The subtyping between parameters is contravariant, compared to the covariant subtyping between signature return types in Rule S-SIG.

The soundness of the judgment is further complicated by the left-to-right matching of the alternations. The judgment

$\boxed{P \not\ll \overline{V}}$

(NP-LINEAR)
$$\frac{m(\overline{x:\tau})\ \overline{V_1} \not\ll' \overline{V_2}}{m(\overline{x:\tau})\ \overline{V_1} \not\ll \overline{V_2}}$$

(NP-SPLIT)
$$\frac{P_1 \not\ll \overline{V_2} \qquad P_2 \not\ll \overline{V_2}}{(P_1 \mid P_2)\ \overline{V_1} \not\ll \overline{V_2}}$$

$\boxed{\overline{V} \not\ll' \overline{V}}$

(NP-DIFF)
$$\frac{m_1 \neq m_2}{m_1(\overline{x:\tau_1})\ \overline{V_1} \not\ll' m_2(\overline{y:\tau_2})\ \overline{V_2}}$$

(NP-VAR)
$$\frac{}{[?*]P\ \overline{V_1} \not\ll' \overline{V_2}}$$

(NP-LONE)
$$\frac{P_1 \not\ll' P_2\ \overline{V_2} \qquad P_1 \not\ll' \overline{V_2}}{P_1 \not\ll' ?P_2\ \overline{V_2}}$$

(NP-END)
$$\frac{}{P \not\ll' \varnothing}$$

(NP-MANY)
$$\frac{P_1 \not\ll' P_2\ *P_2\ \overline{V_2} \qquad P_1 \not\ll' \overline{V_2}}{P_1 \not\ll' *P_2\ \overline{V_2}}$$

(NP-ALT)
$$\frac{P_1 \not\ll' P_2 \qquad P_1 \not\ll' P_3}{P_1 \not\ll' (P_2 \mid P_3)}$$

(NP-VAR/ALT)
$$\frac{}{(P_1 \mid P_2)\ \overline{V_1} \not\ll' \overline{V_2}}$$

$\boxed{\overline{V} \ll \overline{m}}$

(P-LINEAR)
$$\frac{\overline{V} \ll \overline{m}}{m(\overline{x:\tau})\ \overline{V} \ll m\ \overline{m}}$$

(P-END)
$$\frac{}{\overline{V} \ll \varnothing}$$

(P-VAR)
$$\frac{}{[?*]P\ \overline{V} \ll \overline{m}}$$

(P-VAR/ALT)
$$\frac{}{(P_1 \mid P_2)\ \overline{V} \ll \overline{m}}$$

Figure 4. Prefix and negation judgments

a (b | b c) <: a b c should *not* succeed, because requests to a method with the subsequence (b | b c) cannot use the second part of the alternation: it will always be matched by the first. Testing that an earlier prefix cannot have consumed a request is a different question to compatibility, and requires a 'no matching prefix' judgment, defined in Figure 4. The judgment $P \not\ll \overline{V}$ holds if no corresponding part of a valid request to a sequence $\overline{V}$ could have matched the maximal linear prefix of P. Because the start of an alternation can be a further alternation, the judgment must be split into a form that distributes over alternations before testing prefixes.

$\boxed{\Gamma \vdash e : \tau}$

(T-OBJ)
$$\frac{\overline{P \triangleright \overline{x:\tau}} \qquad \Gamma, \mathsf{self} : \mathbf{type}\{\overline{S}\}, \overline{\overline{x:\tau} \vdash e : \tau_r}}{\Gamma \vdash \mathbf{object}\{\overline{\mathbf{method}\ P \to \tau_r\ \{e\}}\} : \mathbf{type}\{\overline{S}\}}$$ (Unique prefixes for $\overline{P}$)

(T-REQ)
$$\frac{\Gamma \vdash e_s : \mathbf{type}\{\overline{S}\} \qquad \max(\{P \to \tau \mid P \to \tau \in \overline{S} \wedge P \ll \overline{m}\}) = P \to \tau_r \qquad \Gamma \vdash P \hookleftarrow \overline{m(\overline{e_p})}}{\Gamma \vdash e_s.\overline{m(\overline{e_p})} : \tau_r}$$

$\boxed{\overline{V} \triangleright \Gamma}$

(M-ONE)
$$\frac{\overline{V} \triangleright \Gamma}{m(\overline{x:\tau})\ \overline{V} \triangleright \overline{x:\tau}, \Gamma}$$

(M-VAR)
$$\frac{P \triangleright \Gamma_1 \qquad \overline{V} \triangleright \Gamma_2}{[?*]P\ \overline{V} \triangleright \mathrm{seq}(\Gamma_1), \Gamma_2}$$

(M-EMPTY)
$$\frac{}{\varnothing \triangleright \varnothing}$$

(M-ALT)
$$\frac{P_1 \triangleright \Gamma_1 \qquad P_2 \triangleright \Gamma_2 \qquad \overline{V} \triangleright \Gamma_3}{(P_1 \mid P_2)\ \overline{V} \triangleright \mathrm{seq}(\Gamma_1), \mathrm{seq}(\Gamma_2), \Gamma_3}$$

$\boxed{\Gamma \vdash P \hookleftarrow \overline{m(\overline{e})}}$

(T-ONE)
$$\frac{\Gamma \vdash e_p : \tau_p \qquad \Gamma \vdash \overline{V} \hookleftarrow \overline{m(\overline{e})}}{\Gamma \vdash m(\overline{x:\tau_p})\ \overline{V} \hookleftarrow m(\overline{e_p})\ \overline{m(\overline{e})}}$$

(T-LONE)
$$\frac{\Gamma \vdash P\ \overline{V} \hookleftarrow \overline{m(\overline{e})}}{\Gamma \vdash ?P\ \overline{V} \hookleftarrow \overline{m(\overline{e})}}$$

(T-SKIP)
$$\frac{P \not\ll \overline{m()} \qquad \Gamma \vdash \overline{V} \hookleftarrow \overline{m(\overline{e})}}{\Gamma \vdash [?*]P\ \overline{V} \hookleftarrow \overline{m(\overline{e})}}$$

(T-MANY)
$$\frac{\Gamma \vdash ?P\ *P\ \overline{V} \hookleftarrow \overline{m(\overline{e})}}{\Gamma \vdash *P\ \overline{V} \hookleftarrow \overline{m(\overline{e})}}$$

(T-EMPTY)
$$\frac{}{\Gamma \vdash \varnothing \hookleftarrow \varnothing}$$

(T-ALT/LEFT)
$$\frac{\Gamma \vdash P_1\ \overline{V} \hookleftarrow \overline{m(\overline{e})}}{\Gamma \vdash (P_1 \mid P_2)\ \overline{V} \hookleftarrow \overline{m(\overline{e})}}$$

(T-ALT/RIGHT)
$$\frac{P_1 \not\ll \overline{m()} \qquad \Gamma \vdash P_2\ \overline{V} \hookleftarrow \overline{m(\overline{e})}}{\Gamma \vdash (P_1 \mid P_2)\ \overline{V} \hookleftarrow \overline{m(\overline{e})}}$$

Figure 5. Partial typing judgment

While soundness w.r.t. request compatibility is achieved in spite of the greedy matching, it is more difficult to achieve completeness. A signature which cannot be requested such as a *b b should require no matching sequence, as its inability to be requested means that request compatibility is unaffected. The provided compatibility rules still require that

$\boxed{e \longmapsto e}$

(R-REQ)
$$\frac{\max(\{P \to \tau\ \{e\} \mid \textbf{method}\ P \to \tau\ \{e\} \in \overline{M} \wedge P \ll \overline{m}\}) = P \to \tau_r\ \{e\} \qquad P \hookleftarrow \overline{m(\overline{O})} \triangleright v}{\textbf{object}\{\overline{M}\}.\overline{m(\overline{O})} \longmapsto e[\text{sub}(v)]}$$

$\boxed{P \hookleftarrow \overline{m(\overline{e})} \triangleright v}$

(R-ONE)
$$\frac{v[\overline{x := e_p}] \qquad \overline{V} \hookleftarrow \overline{m(\overline{e})} \triangleright v_2}{m(\overline{x : \tau})\ \overline{V} \hookleftarrow m(\overline{e_p})\ \overline{m(\overline{e})} \triangleright v_1, v_2}$$

(R-SKIP)
$$\frac{P \not\ll \overline{m()} \qquad \overline{V} \hookleftarrow \overline{m(\overline{e})} \triangleright v}{[?*]P\ \overline{V} \hookleftarrow \overline{m(\overline{e})} \triangleright \text{empty}(P), v}$$

(R-LONE)
$$\frac{P\ \overline{V} \hookleftarrow \overline{m(\overline{e})} \triangleright v}{?P\ \overline{V} \hookleftarrow \overline{m(\overline{e})} \triangleright \text{sing}(P, v)}$$

(R-MANY)
$$\frac{?P\ *P\ \overline{V} \hookleftarrow \overline{m(\overline{e})} \triangleright v}{*P\ \overline{V} \hookleftarrow \overline{m(\overline{e})} \triangleright v}$$

(R-EMPTY)
$$\frac{}{\varnothing \hookleftarrow \varnothing \triangleright \varnothing}$$

(R-ALT/LEFT)
$$\frac{P_1\ \overline{V} \hookleftarrow \overline{m(\overline{e})} \triangleright v}{(P_1 \mid P_2)\ \overline{V} \hookleftarrow \overline{m(\overline{e})} \triangleright \text{sing}(P_1, v), \text{empty}(P_2)}$$

(R-ALT/RIGHT)
$$\frac{P_1 \not\ll \overline{m()} \qquad P_2\ \overline{V} \hookleftarrow \overline{m(\overline{e})} \triangleright v}{(P_1 \mid P_2)\ \overline{V} \hookleftarrow \overline{m(\overline{e})} \triangleright \text{empty}(P_1), \text{sing}(P_2, v)}$$

Figure 6. Partial reduction rules

a 'compatible' set of parts appear in the subsequence, and detecting signatures which cannot be requested is difficult enough that we do not address it here.

6.2 Typing and Reduction

The type rules relevant to the extension are given in Figure 5. The extension has made two changes. Typing of object literals (the only place where methods appear) is now complicated by the possible appearance of variable parts. The judgment $\overline{V} \triangleright \Gamma$ converts a sequence of method parts into a set of type assumptions. We assume the existence of the 'max' auxiliary function, which selects the signature with the longest linear prefix from a set, and the 'seq' auxiliary function, which maps every type in a context into a sequence type (defined as a type with an appropriate do() method, as in the examples).

Requests now need to perform signature lookup by method prefix, choosing the longest if there are multiple results, and then check the types of the arguments in the request parts against the types in the signature's parts. The judgment $\overline{V} \ll \overline{m}$, also defined in Figure 4, is essentially a simpler form of the dual to $\not\ll$, as one of the sides is completely linear. $\Gamma \vdash P \hookleftarrow \overline{m(\overline{e})}$ performs the matching of part names to the linear and variable parts in the signature, checking the types of the arguments against the required types for parameters as it goes.

Again, in order to simulate the greedy matching, we need a concept of no matching prefix judgment. Because the existing judgment does not take parameters into account, stripping the arguments out of the method request and applying it as a linear method signature allows reuse of the judgment.

The relevant reduction rules are provided in Figure 6. The rules for $P \hookleftarrow \overline{m(\overline{e})} \triangleright v$ closely follow the similar type judgment, but have to substitute the actual values in for the parameters instead of checking the types. Because some of the parameters need to be collected and transformed into an iterable object, the traversal over the method parts collects the argument values in a map instead of performing the substitution directly. Concatentation of maps concatenates sequence values when a key is present on both sides. As before, we assume there are informally defined auxiliary functions in the rules: 'empty' builds a map with an empty list for each of the parameter names in the given sequence, 'sing' wraps the corresponding values of the parameter keys in the given part into singleton lists, and 'sub' transforms a map into the substitutions defined by the map, translating lists into sequence objects.

6.3 Properties

We wish to demonstrate that the variable method parts extension preserves an existing proof of soundness for the language's type system. To do this, we show that both progress and preservation are not damaged by the extension. We do not formally discuss the preservation rule, as we need only demonstrate that substitution preservation still works. This follows from a straightforward examination of the new rules, given how the typing and reduction rules essentially map onto each other.

To ensure that the existing progress theorem is undamaged, we only need to show that the method part compatibility judgment is actually sound w.r.t. request compatibility: that if a fully resolved request expression is well-typed, then it can be reduced to the (substituted) body of a method in the object.

Lemma 1. *If* $P_1 <: P_2$ *and* $P_2 \ll \overline{m}$*, then* $P_1 \ll \overline{m}$*.*

Proof. By induction on the derivation of $P_2 \ll \overline{m}$. Both sequences begin with a literal part, so in order for $P_1 <: P_2$ to hold, Rule S-ONE must hold. In subsequent steps through the derivation, subtracting corresponding parts from both sequences, P_1 will meet a variable part, in which case the axioms of $\ll$ apply. $P_1 <: P_2$ guarantees that P_1 cannot remain linear after P_2 has ended or met a variable part. □

Lemma 2.
If $P_1 <: P_2$ *and* $\Gamma \vdash P_2 \hookleftarrow \overline{m(\overline{e})}$*, then* $\Gamma \vdash P_1 \hookleftarrow \overline{m(\overline{e})}$*.*

Proof. By induction on the derivation of $\Gamma \vdash P_2 \hookleftarrow \overline{m(\overline{e})}$ with a case analysis on the last step, matching rules in $P_1 <: P_2$ to find an appropriate rule to advance the goal. S-SKIP might apply at any time, but it proceeds normally and so does not affect the proof. T-ONE is only handled by a literal part, and S-ONE upholds parameter typing through the parameter compatibility relationship and subsumption. T-LONE, T-MANY, and T-SKIP are handled by the lone or many rules, so the compatibility rule may apply or skip, T-EMPTY is handled by a series of skips until P_1 is also empty, and the remaining alternation rules are trivially handled by the inversion of S-ALT. □

Theorem 1. *If* $\Gamma \vdash O_s.\overline{m(\overline{O_p})} : \tau$ *then* $O_s.\overline{m(\overline{O_p})} \longmapsto e'$*.*

Proof. Because $\Gamma \vdash O_s.\overline{m(\overline{O_p})} : \tau$, there exists a signature with method parts P_1 in the type whose prefix matches the request $\overline{m}$, and the method inclusion property of the underlying type system ensures that there is a method M with parts P_2 in the corresponding object such that $P_2 <: P_1$. By Lemma 1, $P_2 \ll \overline{m}$.

It remains to show that $P_2 \hookleftarrow \overline{m(\overline{O_p})} \triangleright v$, with a straightforward application of Lemma 2, and then matching the similar rules from $\Gamma \vdash P_2 \hookleftarrow \overline{m(\overline{O_p})}$ obtained from inversion of the hypothesis. □

7. Related Work

Multi-part Method Names ("selectors") were first introduced in Smalltalk-76 [25], and are widely used in later Smalltalk-derived languages, such as Self [38] and Newspeak [5]. Objective-C [7] introduced multi-part method names into a C-derived syntax, and more recently Elements Oxygene [34] supports multi-part names for Objective-C compatibility. Smalltalk, Self, and Newspeak use multi-part method names for control structures and consequently suffer from the problems of multiple method definitions we address in this paper: we expect these languages could equally benefit from the more flexible method definitions we propose.

In Pharo Smalltalk 4.0, for example, the Collection class has four separate method definitions to cover variants of `detect: ?ifFound: ?ifNone:`, and definitions of class methods `+with:` for up to six arguments. The Block class has four definitions each for `+value:` and `+cull:`. All Objects have four definitions for `perform:*with:`, and ten for variants of `when:(send:|sendOnce:)to:(with:|withArguments:)?exclusive:`. In Self 4.5 blocks have six definitions for `value:*With:`, and strings over twenty for various cases of `sendTo:?DelegatingTo:*With:`.

More generally, linguistic support for optional and named ("keyword") parameters goes back at least as far as Lisp Machine Lisp in the early 1980s [32]. Ada included optional and named parameters in the Pascal tradition at around the same time, and later C++ and C♯ in the C tradition; they have also been proposed recently for Java [18]. Contemporary dynamic languages have either tended to follow Lisp with explicit support for optional named arguments (as in Python and Ruby 2) or rely on passing explicit hashes (Perl, Ruby 1). None of these offer the kind of flexible arguments with optional, repeating, and nested parts that we provide.

Domain Specific Languages. The first organised effort to build and combine small, special purpose, domain specific languages (DSLs) was arguably the many "little languages" written for Unix, also in the early to mid 1980s [2]. More recently, building DSLs has become a recognised development technique: Volker and Fowler provide good overviews of the industrial state of practice [15, 39]. Fowler makes the key distinction between "external" DSLs that are implemented from scratch (either with compiler-compilers such as yacc, or more recently, language workbenches such as SpooFax, Cedalion, or Rascal [13, 28–30]) and "internal" (or "embedded") DSLs that are implemented within a host language. Generalised method names support internal DSLs, so we do not consider external DSLs further here.

Tobin-Hochstadt *et al.* [36] describe languages as libraries in Racket, a Scheme-based language that combines a flexible Lisp Machine Lisp—style function call syntax with a powerful macro language [9]. Racket reintroduced the concept of using multiple "language levels" for teaching [14], originally from SP/k [19], and much of Grace's design is inspired by Racket. In practice, language definition in Racket relies heavily on macros, whereas in Grace we aim to see how far we can get with flexible method names but without macros or defining forms. The advantage of avoiding macros is that we avoid code that does not do what it appears to do: arguments are always evaluated before methods are requested, new bindings are never introduced implicitly, and parse or type errors can stem only from what was actually written in the input source code [20].

SugarJ [11] aims to provide Java with the same kind of extensibility as macros in Racket or Lisp: Java programs can

include "sugar" definitions that define new syntax very flexibly, and then provide rules that translate the sugar into standard Java code. SugarHaskell promises similar features for Haskell [12]. Like other kinds of macros, SugarJ can provide very flexible definitions, at a cost of introducing a whole new kind of definition into the language, and making the evaluation model significantly more complex (first macro-expansion, then evaluation). In contrast, our approach does not change either the caller-side syntax or the evaluation model of Grace, but lets us write more flexible definitions within the existing syntactic and semantic framework.

Ruby domain-specific languages are common [15]. Ruby's open classes permit modifying third-party classes, including built-in objects, to add new globally-visible methods, enabling users of the DSL to write, for example, `3.years .ago` to represent a time. Ruby also supports dynamically-bound block evaluation using the `instance_eval` method, which executes a block of code inside another object and with access to methods defined in that object. Ruby's syntax permits reasonably fluid code to be written in this way, and different DSLs may be used at different points by evaluating code inside different objects. In contrast, we provide DSL definitions without requiring their authors to undertake meta-programming, whether static or dynamic.

DSLs are not the sole preserve of dynamically typed languages. Scala [33, 35] includes several features supporting DSLs, such as methods acting like built-in structures, operators with many levels of precedence and associativity, and implicit parameters that allow arguments to be passed automatically. Like Racket, Scala also includes powerful macro features [6, 10] integrating the compiler and runtime. Haskell is also used to define domain specific languages [1, 26], typically by using the language's type classes to embed them. Static type information directs which functions are actually executed for a particular expression, often based upon the calling context. A programmer can temporarily enter the domain of a DSL simply by declaring the return type of their function. In contrast to Scala and Haskell, neither Grace's original syntax, while flexible, or our extensions, admit ambiguities that need to be resolved by static types. Grace programs have the same semantics with or without type definitions, and no matter how complex our extended multiple-part method definitions, there is no change to the client (caller-side) syntax. The run-time behaviour of dialects is exactly Grace method execution.

Regular-Expression Types. XDuce [24] and ℂDuce [3] are domain specific languages for describing the types of XML documents with a similar set of regular expression combinators to the ones we have presented here: for instance the type `tag[child1 | child2*]`. OCamlDuce [16] is a similar extension to OCaml. All of the XML elements are easily identified by their tag name (which makes matching elements against types easier) but the type matching also backtracks. We have accepted the semantic complication of greedy matching for the aforementioned practical benefits.

The subtyping in XDuce operates in the opposite direction to the signature compatibility provided here, as their regular expressions operate over types whereas our expressions operate over methods. Accounting for more possibilities in XDuce permits more elements to conform to a type, but the same operation on method parts increases the constraints on the types instead. The XDuce type system also features recursive types, whereas there are no recursive references in the method part sequences as it is not considered a practical concern.

8. Future Work

Performance optimisations to this system remain an open question, and incorporating static type information into the process is an area we intend to explore. We hope to apply generalised method names in our work on visual programming tools [21, 22], where they can make families of method variants accessible through the use of only a small number of common sub-parts. As well, the techniques we used here for multi-part names could also be applied within the parameter lists of single-part names in languages without support for multi-part names.

9. Conclusion

Defining families of multi-part method names with repetition, alternation, and optionality enables domain-specific languages and control structures to be defined more concisely and easily, library APIs to act in ways previously restricted to DSLs, and all of these to be checked for correctness both at run time and statically with meaningful error reports.

We have described a design for generalising methods, with minimal restrictions on their content enabling informative error messages, and without any impact on the user-level language, concentrating all complexity on the advanced programmers writing libraries with no change to call-side syntax. We have presented both a fully-featured implementation of these methods and case studies in each of our application domains showing generalised names in practice. Further, we showed that families of method names defined in this way do not affect type soundness.

Our original goal for generalised names was only to simplify the definitions of control structures, but we found applications in other areas that overwhelmed the importance of our original goal. In particular, allowing ordinary library APIs to define elegant, enforceable, extensible, encapsulated sublanguages for interacting with them without any end-user impact was an unanticipated benefit sufficient to recommend the introduction of generalised names into any language with multi-part methods. Generalised names turn out to be a relatively small extension to an object-oriented language providing a disproportionate amount of power to programmers.

References

[1] AUGUSTSSON, L., MANSELL, H., AND SITTAMPALAM, G. Paradise: a two-stage DSL embedded in Haskell. In *ICFP* (2008), pp. 225–228.

[2] BENTLEY, J. Programming pearls: Little languages. *CACM 29*, 8 (Aug. 1986), 711–721.

[3] BENZAKEN, V., CASTAGNA, G., AND FRISCH, A. CDuce: An XML-centric general-purpose language. In *ICFP* (2003), pp. 51–63.

[4] BLACK, A. P., BRUCE, K. B., HOMER, M., AND NOBLE, J. Grace: the absence of (inessential) difficulty. In *Onward!* (2012), pp. 85–98.

[5] BRACHA, G., VON DER AHÉ, P., BYKOV, V., KASHAI, Y., MADDOX, W., AND MIRANDA6, E. Modules as objects in Newspeak. In *ECOOP* (2010).

[6] BURMAKO, E., ODERSKY, M., VOGT, C., ZEIGER, S., AND MOORS, A. Scala macros. `http://scalamacros.org`, Apr. 2012.

[7] COX, B. *Object Oriented Programming: An Evolutionary Approach*. Addison Wesley, 1991.

[8] CUCUMBER LTD. Cucumber. `https://cucumber.io/`.

[9] CULPEPPER, R., TOBIN-HOCHSTADT, S., AND FLATT, M. Advanced macrology and the implementation of Typed Scheme. In *ICFP workshop on Scheme and Functional Programming* (2007).

[10] EPFL. Environment, universes, and mirrors - Scala documentation. `http://docs.scala-lang.org/overviews/reflection/environment-universes-mirrors.html`, 2013.

[11] ERDWEG, S., RENDEL, T., KÄSTNER, C., AND OSTERMANN, K. SugarJ: Library-based syntactic language extensibility. In *OOPSLA* (2011).

[12] ERDWEG, S., RIEGER, F., RENDEL, T., AND OSTERMANN, K. Layout-sensitive language extensibility with SugarHaskell. In *HASKELL* (2012), pp. 149–160.

[13] ERDWEG, S., VAN DER STORM, T., VÖLTER, M., BOERSMA, M., BOSMAN, R., COOK, W., GERRITSEN, A., HULSHOUT, A., KELLY, S., LOH, A., KONAT, G., MOLINA, P., PALATNIK, M., POHJONEN, R., SCHINDLER, E., SCHINDLER, K., SOLMI, R., VERGU, V., VISSER, E., VAN DER VLIST, K., WACHSMUTH, G., AND VAN DER WONING, J. The state of the art in language workbenches. In *Software Language Engineering*, vol. 8225 of *Lecture Notes in Computer Science*. Springer International Publishing, 2013.

[14] FINDLER, R. B., CLEMENTS, J., FLANAGAN, C., FLATT, M., KRISHNAMURTHI, S., STECKLER, P., AND FELLEISEN, M. DrScheme: a programming environment for Scheme. *J. Funct. Program. 12*, 2 (3 2002), 159–182.

[15] FOWLER, M. *Domain Specific Languages*. Addison-Wesley, 2011.

[16] FRISCH, A. OCaml + XDuce. In *ICFP* (2006), pp. 192–200.

[17] GAMMA, E., HELM, R., JOHNSON, R. E., AND VLISSIDES, J. *Design Patterns*. Addison-Wesley, 1994.

[18] GIL, J. Y., AND LENZ, K. Keyword- and default- parameters in Java. *Journal of Object Technology 11*, 1 (2011), 1:1–17.

[19] HOLT, R. C., AND WORTMAN, D. B. A sequence of structured subsets of PL/I. *SIGCSE Bull. 6*, 1 (Jan. 1974), 129–132.

[20] HOMER, M., JONES, T., NOBLE, J., BRUCE, K. B., AND BLACK, A. P. Graceful dialects. In *ECOOP* (2014), pp. 131–156.

[21] HOMER, M., AND NOBLE, J. A tile-based editor for a textual programming language. In *Proceedings of IEEE Working Conference on Software Visualization* (Sept 2013), VISSOFT'13, pp. 1–4.

[22] HOMER, M., AND NOBLE, J. Combining tiled and textual views of code. In *Proceedings of IEEE Working Conference on Software Visualization* (Sept 2014), VISSOFT'14.

[23] HOMER, M., NOBLE, J., BRUCE, K. B., BLACK, A. P. ., AND PEARCE, D. J. Patterns as objects in Grace. In *DLS* (2012), pp. 17–28.

[24] HOSOYA, H., VOUILLON, J., AND PIERCE, B. C. Regular expression types for XML. In *ICFP* (2000), pp. 11–22.

[25] INGALLS, D. H. H. The Smalltalk-76 programming system design and implementation. In *POPL* (1978).

[26] JONES, M. P. Experience report: playing the DSL card. In *ICFP* (2008).

[27] JONES, T., AND NOBLE, J. Tinygrace: A simple, safe and structurally typed language. In *FTFJP* (2014).

[28] KATS, L. C., AND VISSER, E. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA* (2010).

[29] KLINT, P., VAN DER STORM, T., AND VINJU, J. Rascal: A domain specific language for source code analysis and manipulation. In *SCAM* (2009), pp. 168–177.

[30] LORENZ, D. H., AND ROSENAN, B. Cedalion: a language for language oriented programming. In *OOPSLA* (Oct. 2011).

[31] MEIJER, E., BECKMAN, B., AND BIERMAN, G. LINQ: Reconciling object, relations and XML in the .NET framework. In *SIGMOD* (2006), pp. 706–706.

[32] MOON, D., STALLMAN, R., AND WEINREB, D. *The Lisp Machine Manual*, third ed. MIT AI Lab, 1981.

[33] ODERSKY, M. The Scala language specification. Tech. rep., Programming Methods Laboratory, EPFL, 2011.

[34] REMOBJECTS SOFTWARE. The Oxygene language. `elementscompiler.com/elements/oxygene/`, 2015.

[35] ROMPF, T., AND ODERSKY, M. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE* (2010), pp. 127–136.

[36] TOBIN-HOCHSTADT, S., ST-AMOUR, V., CULPEPPER, R., FLATT, M., AND FELLEISEN, M. Languages as libraries. In *PLDI* (2011).

[37] TORGERSEN, M. Querying in C#: How language integrated query (LINQ) works. In *OOPSLA Companion* (2007).

[38] UNGAR, D., AND SMITH, R. B. SELF: the Power of Simplicity. *Lisp and Symbolic Computation 4*, 3 (June 1991).

[39] VOELTER, M. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. `dslbook.org`, 2013.