Dynamic Operator Overloading in a Statically Typed Language

Olivier L. Clerc and Felix O. Friedrich

Computer Systems Institute, ETH Zürich, Switzerland
`olivier.clerc@alumni.ethz.ch`, `felix.friedrich@inf.ethz.ch`

October 31, 2011

Abstract

Dynamic operator overloading provides a means to declare operators that are dispatched according to the runtime types of the operands. It allows to formulate abstract algorithms operating on user-defined data types using an algebraic notation, as it is typically found in mathematical languages.

We present the design and implementation of a dynamic operator overloading mechanism in a statically-typed object-oriented programming language. Our approach allows operator declarations to be loaded dynamically into a running system, at any time. We provide semantical rules that not only ensure compile-time type safety, but also facilitate the implementation. The spatial requirements of our approach scale well with a large number of types, because we use an adaptive runtime system that only stores dispatch information for type combinations that were encountered previously at runtime. On average, dispatching can be performed in constant time.

1 Introduction

Almost all programming languages have a built-in set of operators, such as `+`, `-`, `*` or `/`, that perform primitive arithmetic operations on basic data types. *Operator overloading* is a feature that allows the programmer to redefine the semantics of such operators in the context of custom data types. For that purpose, a set of operator implementations distinguished by their signatures has to be declared. Accordingly, each operator call on one or more custom-typed arguments will be dispatched to one of the implementations whose signature matches the operator's name and actual operand types.

For certain target domains, operator overloading allows to express algorithms in a more natural form, particularly when dealing with mathematical objects. Overloading also adds a level of abstraction, as it makes it possible to refer to a specific operation by the same symbol, irrespective of the data type. For some matrix-valued variables, the mathematical expression `-(a + b * c)` is obviously much more natural than some nested function calls `MatrixNegation(MatrixSum(a, MatrixProduct(b, c))` or chained method calls `a.Plus(b.Times(c)).Negative()`, as some object-oriented languages allow.

1.1 Static vs. Dynamic Operator Overloading

In statically-typed languages, operators are typically also dispatched in a static fashion, which is also known as ad-hoc polymorphism. That is, the static types of the arguments determine which implementation is to be called [1]. Since all of the static types are resolved by the compiler, the whole operation can be performed at compile-time without any impact on the runtime performance.

Whereas static overloading can be seen as a mere naming convenience, *dynamic operator overloading* truly adds new expressivity to a language. Dynamically overloaded operators are a special case of what is also known as *multi-methods*, i.e., methods that are dispatched with respect to the runtime types of multiple arguments. For example, when the expression `(a * b) / (c * d)` is evaluated, it may depend on the concrete runtime types the two products evaluate to, what implementation is selected to perform the division. Because a dynamic dispatch generally has to be deferred until runtime, it consequently introduces an additional runtime overhead.

1.2 Motivation

Late-binding of operators opens the door for new optimizations. One can express how an operation is performed in general, and then provide additional, more efficient implementations for data of certain subtypes. Naturally, this can also be achieved for operations on single arguments by using ordinary virtual methods, which are overridden by the relevant subtypes. However, methods are not suitable to refine operations on multiple arguments, such as a multiplication.

For instance, the `*` operator may be used to define a general matrix multiplication on the type pair $\langle \texttt{Matrix}, \texttt{Matrix} \rangle$. With the introduction of a subtype `SparseMatrix` that represents sparse matrices [2], more efficient implementations may be provided for three different scenarios:

1. The case where the sparse matrix is multiplied from the left hand side: $\langle \texttt{SparseMatrix}, \texttt{Matrix} \rangle$.
2. The same for the right hand side: $\langle \texttt{Matrix}, \texttt{SparseMatrix} \rangle$.
3. And also, when both operands are known to be sparse: $\langle \texttt{SparseMatrix}, \texttt{SparseMatrix} \rangle$.

A possible application of this is a software library for linear algebra in which mathematical objects, such as vectors, matrices and tensors,

[1] The static return type might possibly also be included.
[2] That is, matrices that contain mostly zeros.

are modeled as object types. Algorithms on those object types could be written in an abstract form using ordinary mathematical notation. The same algorithms would then automatically be executed differently, depending on the type of data they are applied on.

In our preceding work, the need for dynamic operators originated during the development of so called Array-Structured Object Types (cf. [5]), which are a special kind of object types that implement the interface of an array. For instance, they allow to implement a sparse matrix, such that it can be referred to as a regular 2-dimensional array in spite of the fact that a compressed storage scheme is used. However, as soon as an instance of such a type is passed to a procedure that accepts any array, only the dynamic type of the parameter tells its dedicated storage scheme apart from a regular one. Without looking at the runtime type, operations on such objects can only access the data elements through the general array interface, i.e., sparse matrices would have to be treated as normal matrices when arithmetic operations are performed on them.

1.3 Our Vision

The goal of this work was to integrate dynamic operator overloading into an object-oriented and statically-typed language without abandoning type-safty. We wanted to have a clear and coherent language design that allows static and dynamic operators to coexist. Both operands of a binary operator should be treated equally, which implies that binary operators should not be members of either of the two object types [3].

Runtime errors due to operator overloading should be ruled out by a strict set of semantical rules to be enforced by the compiler. The return types of operators should be forced to be non-contradictory, such that each operator call is handled by an implementation that returns a compatible value.

As with multi-methods, an operator call should be dispatched to the implementation that has the highest specificity. This measure should be defined according to a clearly-defined concept of type-distance.

Furthermore, the runtime overhead of the dynamic dispatch should not compromise the performance of a statically-typed compiled language. (1) The implementation should be time-efficient, i.e., perform a dynamic double dispatch in amortized constant time. Morever, (2) it should also be space-efficient, which excludes compiler-generated lookup-tables for all possible type combinations [4].

In addition to that, (3) we wanted our implementation to support dynamic module loading. That is, it should be possible to load new modules containing additional operator declarations into a running system such that the new sub-operators are immediately incorporated, without recompilation of the existing modules.

[3] Conceptually, binary operators reside in the Cartesian product of two types.
[4] Unless some form of compression is adopted.

1.4 Related Work

The Common Lisp language is probably the most famous example of a language whose object system natively supports multi-methods [3].

The Python language does not have this feature, however, multi-dispatching capabilities can be added by means of libraries [7].

In Microsoft's C# language, dynamic overloading capabilities were introduced with the advent of a special `dynamic` type that acts as a placeholder for types that are only resolved at runtime [2]. However, since static type checking is bypassed for expressions of this type, no guarantees can be given for a dynamic operator call as to whether it will be handled successfully. The same applies to the type `id` of the Objective-C language [1].

A sophisticated mechanism for dynamic multi-dispatching is presented in [4], which is both time- and space-efficient. In this approach, the compiler generates a lookup automaton that takes the sequence of runtime parameter types one after the other as input. A limitation of this approach is that the generated transition-arrays, which represent the automaton, have to be regenerated from scratch if a new multi-method (e.g., dynamic operator) or subtype is introduced. Therefore, this approach is not suitable for a system in which modules containing new types and operators are added incrementally by means of separate compilation and dynamic module loading.

In [8] and [9], a language called Delta is presented that accommodates a multitude of dynamic features, amongst others dynamic operator overloading. In this language overloaded operators constitute normal members of an object type. The operator call `a x b` is evaluated as a member function call `a.x(b)`. Whereas `a`'s dynamic type is automatically incorporated in the dispatch to the corresponding member function, the type of `b` is not. A second dispatch on the type of this parameter has to be programmed manually within the function. In [8] a Delta-specific design pattern is provided, which achieves this. It requires the programmer to list all possible right operand types along with an implementation in a table. The problem with this is that operator calls can only be handled if the right operand type matches exactly with one that was listed in the table.

2 Background

2.1 Subtype Polymorphism

In an object-oriented environment, operator overloading should respect subtype polymorphism. That is, the fact that an instance of a type can be also treated the same way as the ones of the type it was derived from. In the context of operators, this means that an operator implementation should not only be applicable to objects of the types it was directly declared for, but also to any of their subtypes. For example, an operator defined on the type `Matrix` should also be applicable to instances of the subtype `SparseMatrix`. As a consequence, there is generally more than one operator declaration whose signature is compatible to some actual operands. This ambiguity can be seen in the motivational example in Section 1.2. Due to subtype polymorphism, any of the provided operator implementations could handle a multiplication of two sparse matrices. However, an implementation that is defined defined for the type pair $\langle$`SparseMatrix`, `SparseMatrix`$\rangle$ is the most preferable one.

2.2 Sub-Operator Relationship

Similar to the sub-type relation on single object types, a sub-operator relationship can be established between operators. A sub-operator provides the implementation of an operation in a more special case than its super-operators do. Formally, we define an operator O_1 to be a direct sub-operator of O_2 if, and only if, both share the same name and if one of O_1's operand types is a direct subtype of its counterpart in O_2. Accordingly, O_2 then is a super-operator of O_1. The sub-operator relationship is transitive and defines a partial order on the set of operators. Note that the sub-operator relation between two equally named unary operators directly corresponds to the subtype relation of the operands.

Let L and R be some object types, L' and R' their direct subtypes, and finally, L" and R" some second order descendants. Furthermore, let us assume that there are no other types in the system. For a binary operator $\times$ that is defined on the pair $\langle L, R\rangle$, the graph in Figure 1 contains a node for all possible sub-operators.

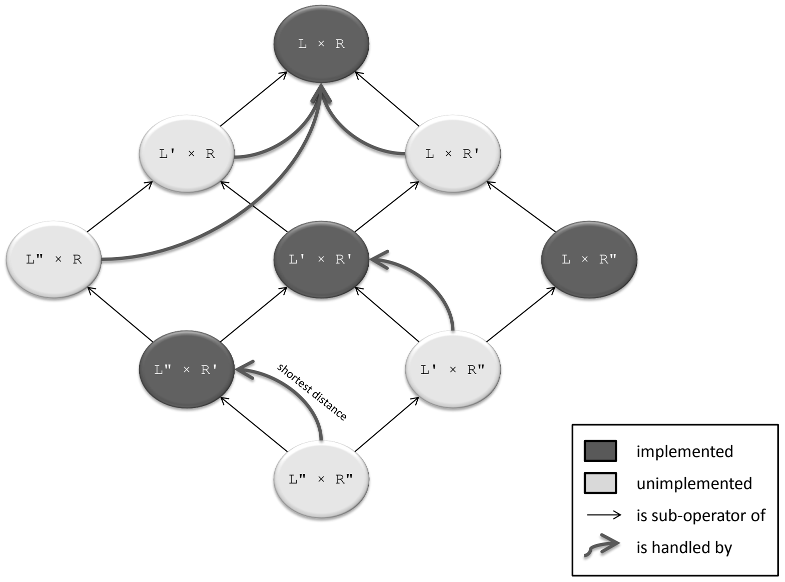

Figure 1: *Sub-operator graph for an operator $\times$ on $\langle L, R\rangle$.*

In this graph, a directed edge is present if the first node represents a direct sub-operator of the second one.

2.3 Type-Distance and Specificity

A concept of type-distance between operators can established to express their similarity. More precisely, we define the type-distance of two operators as the number of direct sub-operator relations between them. This corresponds to the length of the path between their nodes along the directed edges in the sub-operator graph. E.g., the distance between $\langle L", R" \rangle$ and $\langle L, R \rangle$ in Figure 1 is 4. Unrelated operators, such as $\langle L", R' \rangle$ and $\langle L, R" \rangle$, have an infinite distance.

In particular, the measure of type-distance can be used to describe the *specificity* of an operator declaration with respect to a concrete operator call. That is, how closely the declaration's signature matches the call. However, on its own, it is not suitable to determine which operator declaration should be picked, as there generally is more than one declaration at the shortest type-distance. For instance, the nodes $\langle L', R' \rangle$ and $\langle L, R" \rangle$ both have the shortest type-distance to $\langle L', R" \rangle$. In order to resolve this ambiguity, we decided that the operator whose first formal operand has a more concrete type should be given precedence. According to this, $\langle L', R' \rangle$ would have a higher specificity than $\langle L, R" \rangle$, because L' is a subtype of L.

3 The Language

We demonstrate our concept on the basis of a language named Math Oberon (cf. [6]), which is a mathematical extension of Active Oberon and thus a descendant of the language Pascal.

3.1 Declaration of Operators

An implementation of an operator for some formal operand types is provided by the programmer in an operator declaration, which looks much like the definition of a procedure. Listing 1 shows the exemplary module `M` containing two declarations of this kind [5].

```
module M;
type
  Super* = object ... end Super;
  Middle* = object(Super) ... end Middle;

  (* M.+ [1] *)
  operator "+"*(left, right: Super): Super;
  begin ...
  end "+";

  (* M.+ [2] *)
  operator {dynamic} "+"*(left, right: Middle): Middle;
  begin ...
  end "+";
end M.
```

Listing 1: *Module M.*

[5]The asterisks in the code have the effect that the declared types and operators are accessible from other modules.

Note that the actual code of the implementations was omitted. Because the object type `Middle` is defined as a subtype of `Super`, the first operator declaration defines a super-operator of the second one.

Listing 2 illustrates a second module called `N`, which imports `M`. Additionally, `Sub` is introduced as a subtype of `Middle`, along with two new operators that are defined on this type. Both of them are sub-operators of the ones that are imported from the other module.

```
module N;
import M;
type
  Sub* = object(M.Middle) ... end Sub;

  (* N.+ [1] *)
  operator {dynamic} "+"*(left: Sub; right: M.Middle): M.Middle;
  begin ...
  end "+";

  (* N.+ [2] *)
  operator {dynamic} "+"*(left, right: Sub): Sub;
  begin ...
  end "+";

  ...
var
  a, b, c: M.Middle
begin
  a := ...;
  b := ...;
  c := a + b
end N.
```

Listing 2: *Module N.*

By default, operators are overloaded statically, as it would be the case for the first one declared in module `M` (cf. Listing 1). The presence of the `dynamic` modifier instructs the compiler that dispatching is performed dynamically.

3.1.1 Scope of Operators

In our design, an operator declaration does neither reside in a global scope nor in the scope of one of its operand types. Instead, operators belong to the scope of a module that...

- has access to the types of both operands;
- contains the declaration of one of the two formal operand types.

For some formal operand types, there is at most one module that fulfills the two conditions in a given import hierarchy. Note that there are constellations that prohibit operators on certain type combinations.

3.1.2 Co-variance

The return types that the programmer specifies do not affect how an operator is dispatched. The reason is that they cannot be incorporated in a dynamic dispatch, as this would require access to a result's runtime type, in advance. However, in our design, return types have to be *co-variant* with the operand types. For instance, a sub-operator can only return a type that is compatible to the ones being returned by all of it super-operators. Moreover, the absence of any return type in an operator prevents all of its sub-operators from having one.

Return type co-variance ensures type-safety, and forces the programmer to define operators in a coherent and non-contradictory fashion. For instance, the operators declared in module `N` (cf. Listing 2) must return instances of `Middle` or subtypes thereof, as it has already been established by the second declaration in module `M`.

In addition to that, the property of being dynamic also has to be co-variant with the sub-operator relation. A dynamic operator can be seen as special case of a static operator. Hence, a static operator is allowed to have both static and dynamic sub-operators. However, all sub-operators of a dynamic operator must be dynamic. As a consequence, for any dynamic operator, all of its loaded sub-operators are considered for a dynamic dispatch.

3.2 Usage of Operators

The programmer may use an operator, i.e., apply it on some operands, if there is a *reference operator declaration* for the scenario. That is, there must be a declared operator with the same name that is defined on the static types of the operands or supertypes thereof. Out of all declarations that fulfill this condition the one with the highest specificity (according to the definition in Section 2.3) is considered to be the reference.

The reference declaration also determines whether the call will be handled dynamically or not. For instance, the operator call `a + b` near the end of module `N` (cf. Listing 2) is valid, because module `M` contains a `+` operator declaration defined on pairs of `Middle`, which acts as the reference. As dictated by this declaration, the call will be handled dynamically.

4 Implementation

Our implementation relies on a runtime system (or runtime for short) that keeps track of the mappings between operator signatures and implementations. Internally, the runtime uses a hash table to map a numerical representation of a signature to the address of the associated implementation. The signature of a binary operator is comprised of an operator name and the two operand types. By using a special pseudotype that marks the absence of any type, unary operators can also be represented as binary ones. As there are currently no operators supported in Math Oberon with more than two operands, only the case of dynamic binary operators had to be implemented. The hash h of an operator's signature is calculated by bitwise shifting ($\ll$) and xoring ($\oplus$) the numerical values that are assigned to the name and operand types $T(\cdot)$.

$$h(name, T(left), T(right)) = name \oplus (T(left) \ll n) \oplus (T(right) \ll m)$$

In order to get a numerical representation for an object type, the type descriptor's address is used. Different shift amounts for n and m ensure that operators on the type pair $\langle A, B\rangle$ are not being assigned the same hash as for $\langle B, A\rangle$. In theory, this method allows to calculate a hash for an arbitrary amount of operands. The pseudotype that is used on the right hand side for unary operators, always evaluates to 0.

An important principle of our approach is that the runtime does not contain an exhaustive list of all possible call scenarios. Instead, the table of mappings starts out as being empty, and only contains entries for the scenarios that have been registered, so far.

4.1 Operator Registration

The runtime provides the procedure `OperatorRuntime.Register(...)` to register an operator implementation under a certain signature. In order that all of the declared operators in a module are registered, the compiler performs some code instrumentalization at the beginning of the module's body. For each operator declaration in the module's scope that happens to be dynamic, a call to the above-mentioned procedure is inserted that registers the declared implementation precisely under the signature specified in the declaration. Listing 3 depicts in pseudocode how this would look like for the two modules in Listing 1 and 2. Note that there is no such call for topmost declaration in module `M`, as it is static. The body of a module is a piece of code that is automatically executed as soon as the module is loaded for the first time. Therefore, operators are not registered before load-time.

```
module M
import OperatorRuntime;
...
begin
 OperatorRuntime.Register("+", <Middle>, <Middle>, <AddressOf(M.+ [2])>);
  ...
end M.

module N;
import M, OperatorRuntime;
...
begin
 OperatorRuntime.Register("+", <Sub>, <Middle>, <AddressOf(N.+ [1])>);
 OperatorRuntime.Register("+", <Sub>, <Sub>, <AddressOf(N.+ [2])>);
  ...
end N.
```

Listing 3: *The bodies of modules M and N after instrumentalization.*

4.2 Operator Selection

In order to dispatch a dynamic operator call to the operator implementation with the highest specificity, the runtime is consulted. More precisely, the procedure `OperatorRuntime.Select(...)`, which the runtime provides, is used to look-up the implementation that is associated with a certain call scenario. For each operator call whose reference declaration happens to be dynamic, the compiler emits a call to this procedure, with a subsequent call to the address that is returned. For instance, the dynamic call `c := a + b`, at the end of module `N` in Listing 2, would conceptually be handled as illustrated in the pseudocode of Listing 4.

```
implementation := OperatorRuntime.Select("+", a, <Middle>, b, <Middle>);
c := implementation(a, b)
```

Listing 4: *Handling of a dynamic operator call using runtime system.*

Note that it is also necessary to pass the static types to the selection procedure. This information is required to handle the case of `NIL` values, for which we decided that the selection should resort to the static type.

Inside the selection procedure, there are *two* cases that can occur:

1. There is an entry in the hash table that exactly matches the call scenario. In this case, the selection procedure simply returns the address of the implementation.
2. There is none. The runtime then has to conduct a breadth-first search in the space of super-operators in order to find a suitable implementation among the already registered ones. This polymorphic search should result in the implementation that has the highest specificity with respect to the call (cf. Section 2.3).

The graph in Figure 1, illustrate an exemplary state of the runtime system. The dark nodes indicate call scenarios for which an implementation already has been registered, whereas bright nodes represent the ones that do not yet have an entry in the hash table. The curved arrows in the same figure show for all unimplemented nodes, which node the polymorphic search would result in. Note that the existence of at least one compatible implementation is guaranteed, since each operator call must be accompanied by a reference declaration.

After the implementation is found, it is registered under the signature that precisely matches the actual operator call. This means that there will be an additional signature in the hash table that is mapped to the implementation.

In the example presented in Listing 1 and 2, the first call on the type pair $\langle \texttt{Middle}, \texttt{Sub} \rangle$ would result in a polymorphic search that would yield the implementation declared on the pair $\langle \texttt{Middle}, \texttt{Middle} \rangle$.

The polymorphic search, being a more costly operation, will not have a substantial impact on the performance, as it is never repeated for a scenario. Each subsequent operator call with the same operand types will be immediately handled by the hash lookup. On average, the runtime performance of the double dispatch remains constant.

5 Conclusion

We presented the syntax and semantics of a language that integrates both static and dynamic operator overloading in a statically-typed object-oriented programming environment. In this design, operators are not members of one of its operand types, but reside in the scope of a clearly-determined module, instead. The semantical rules that are enforced by the compiler, such as co-variance of return types with the sub-operator relationship, or the existence of reference declarations, prevent runtime errors due to operator overloading from occurring.

The mechanism that we propose for dynamic multi-dispatching is based on a runtime system, which maintains a table that maps operator signatures to operator implementations. By using code instrumentalization, all the operators declared by the programmer are registered as soon as the owner module is loaded. A dynamic dispatch is either handled directly by a table lookup, or in the other case, requires a search that yields the implementation with the highest specificity among the already registered ones. The resulting implementation is immediately registered under the signature of the new actual operand types. Hence, each subsequent operator call on the same argument types will not require a computation-intensive search for an implementation.

- On average, a dynamic multi-dispatch can be handled by a simple table look-up, i.e., in constant time.
- The required space is only in the order of relevant type scenarios. Thus, the memory usage of our approach scales very well with a large number of object types, compared to precomputed look-up tables for all possible operand type combinations.
- By means of dynamic module loading, new types and operators can be added incrementally, such that they are immediately incorporated into a running system, without recompilation of the existing code.

References

[1] *The Objective-C Programming Language - Tools Languages: Objective-C.* Apple, Inc., Dezember 2010. Available electronically from: http://developer.apple.com/resources/.

[2] Albahari, J., and Albahari, B. *C# In a Nutshell - The Definitive Reference.* O'Reilly Media, 2010.

[3] Bobrow, D. G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. Commonloops: merging Lisp and object-oriented programming. In *Conference proceedings on Object-oriented programming systems, languages and applications* (New York, NY, USA, 1986), OOPLSA '86, ACM, pp. 17–29.

[4] Chen, W., and Aberer, K. Efficient multiple dispatching using nested transition-arrays. *Arbeitspapiere der GMD No. 906, Sankt Augustin* (1995).

[5] Clerc, O. L. *Array-Structured Object Types in Active Oberon.* Master Thesis, ETH Zürich, 2010.

[6] Friedrich, F., Gutknecht, J., Morozov, O., and Hunziker, P. A mathematical programming language extension for multilinear algebra. In *Proc. Kolloqium über Programmiersprachen und Grundlagen der Programmierung, Timmendorfer Strand* (2007).

[7] Mertz, D. Advanced OOP: Multimethods. *O'Reilly - LAMP: The Open Source Web Platform* (2003). Available electronically from: http://onlamp.com/pub/a/python/2003/05/29/multimethods.html.

[8] Savidis, A. More dynamic imperative languages. *SIGPLAN Not. 40* (December 2005), 6–13.

[9] Savidis, A. Dynamic imperative languages for runtime extensible semantics and polymorphic meta-programming. In *Rapid Integration of Software Engineering Techniques*, N. Guelfi and A. Savidis, Eds., vol. 3943 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 113–128.