Metaphoric Polymorphism: Taking Code Reuse One Step Further

Ran Rinat[1] and Menachem Magidor[2]

[1] Institute of Computer Science, Hebrew University, Jerusalem 91904, Israel
E-mail: rinat@cs.huji.ac.il
[2] Institute of Mathematics, Hebrew University, Jerusalem 91904, Israel
E-mail: menachem@sunset.huji.ac.il

Abstract. We propose two new constructs for object oriented programming that significantly increase polymorphism. Consequently, code may be reused in ways unaccounted for by existing machinery. These constructs of *type correspondence* and *partial inheritance* are motivated from *metaphors* of natural language and thought. They establish correspondences between types non of which is (necessarily) a subtype of the other. As a result, methods may operate on objects - and may receive arguments - of types different than the ones originally intended for. The semantics of the proposed constructs generalizes that of ordinary inheritance, thereby establishing the latter as a special case. We show that the incorporation of these constructs in programming supports the process of natural software evolution and contributes to a better conceptual organization of the type system.

1 Introduction

One important benefit of object oriented programming is the ability to reflect conceptual structure in software: the class hierarchy of an object oriented system represents to a large extent the categorical hierarchy of the real-life domain in which it operates. Some of the most important constructs of object oriented programming are counterparts of mechanisms that exist in human cognitive activity. For example, inheritance reflects the **is-a** relationship between concepts ([2], [15], [14]), and genericity represents what may be called "parameterized concepts".

Imitating cognitive mechanisms leads to a better organization of software, resulting in a higher degree of *code reuse*. In particular, because classes are organized in an inheritance hierarchy, a single piece of code may apply to a whole sub-hierarchy of classes. With genericity, types that have common properties and behavior may be derived from a single generic class, allowing them to share the code for that class. This can also be conceived as a sort of organization method, imitating human conceptual capabilities.

The (human) conceptual system, however, has a much more complex structure than what is captured by existing object oriented constructs. This structure is analyzed by George Lakoff in his book [6]. One factor that is singled out there as being central in human categorization is that of *metaphors*. In an earlier

book by the same author and by Mark Johnson ([7]), which is wholly dedicated to the issue of metaphors, it is argued that "the human conceptual system is metaphorically structured and defined". The authors demonstrate through numerous examples, that metaphors are not (just) elements of poetry, but are constantly used in everyday speech, affecting the way in which people perceive, think and act. Quoting from [7], "The essence of metaphor is understanding and experiencing one thing in terms of another". We bring their first example to illustrate what is meant by that: consider the metaphor ARGUMENT IS WAR. This metaphor is reflected in everyday language by many expressions:

- Your claims are *indefensible.*
- She *attacked every weak point* in my argument.
- His criticisms were *right on target.*
- If you use that *strategy*, he'll *wipe you out.*
- I've never *won* an argument with her.

As seen by this example, notions from the realm of war are applied to arguments. However, an argument is not *really* a war, in the sense that only *part* of what constitutes the concept *war* is applicable to arguments. Put it another way, there is a *partial mapping* from the building blocks of *war* to those of *argument.*

Looking at this argumentation, and considering the benefit that has been gained from incorporating conceptual oriented constructs into programming, it seems reasonable that metaphors could motivate some useful programming constructs.

We propose two new constructs for object oriented programming: *type correspondences* and *partial inheritance.* Both of them are based on the observation that a piece of code, originally written to work with certain types, can actually work with other ones, which are not necessarily subtype compatible. The idea, motivated by metaphors, is to *partially* relate two types in a useful way.

To illustrate, assume we have a class *INTEGER* with methods *multiply* (n:*INTEGER*): *INTEGER* and *power* (n:*INTEGER*): *INTEGER.* The first returns the result of multiplying the receiving object by n, and the second returns the result of raising it to the power of n. We also have a class *MATRIX* of square matrices of a given fixed size, equipped with a method *matrix_multiply (some_matrix:MATRIX): MATRIX* which returns the result of multiplying the target object by *some_matrix.* Our goal is to make use of the fact that the power function on matrices relates to matrix multiplication just as the power function on integers relates to integer multiplication, in order to be able to apply *power* to matrices. This is what metaphors do: they use similarities between concepts to apply notions of one to the other.

To account for that we define a *type correspondence* that relates the types *INTEGER* and *MATRIX*, mapping *multiply* of *INTEGER* to *matrix_multiply* of *MATRIX.* Having done that, and assuming that *power* is implemented using *multiply* (but not in any specific manner), we may now apply *power* to objects of type *MATRIX.* That is, we may issue a call matrix.*power*(n), where matrix is a (variable of type) *MATRIX* and n is an integer. It will work, because whenever

multiply is mentioned in *power*'s text, *matrix_multiply* will be invoked instead, since *multiply* has been mapped to it in the type correspondence. Thus, the code for *power*, originally written for integers, has been reused for *MATRIX*, which is not a subtype of *INTEGER*[3]. Note that *INTEGER* may have other attributes and methods not mapped to *MATRIX*, such as *prime:BOOLEAN*, and so the mapping is partial in that sense. The proposed syntax for type correspondences, that also declares *power* as applied to *MATRIX*, is given in Sect. 2.1.

In this examples, a new working method has been added to the type *MATRIX*. But the main point is not so much this enrichment of *MATRIX*'s set of methods, as it is the use of a certain piece of code (that of *power*) with argument types not originally intended for. *power* actually has two arguments: **self** and n, both *intended* to be integers. Our type correspondence makes the text of *power* meaningful when the first argument (**self**) is a matrix and the second still an integer. That *power* is also added to the list of methods supported by *MATRIX* so as to legalize calls such as matrix.*power*(n) is another separate feature offered by the construct of type correspondence. This becomes clearer if we take *power* to be a free-standing function rather than a method of *INTEGER*. In this case *power(k,m : INTEGER) : INTEGER* receives two (intended) integers k and m and returns k^n. Given the type correspondence relating *INTEGER* to *MATRIX*, we could supply a matrix argument for k, yielding a matrix result.

The second construct we propose is *partial inheritance*. It is conceptually the same as type correspondences, only it establishes the (partial) correspondence while actually creating one of the classes.

The *INTEGER-MATRIX* case is a toy example that involves only one type correspondence. However, in realistic cases, a set of type correspondences will be needed in favor of a single reuse task, involving a collection of collaborating classes. In fact, when partial inheritance is also employed, we conjecture that frameworks, that is libraries of collaborating classes, could be reused in new and unanticipated ways. Although we do not show a framework example in this paper, a complicated enough scenario is outlined in Sect. 3.2, which we believe testifies to that effect.

The semantics of the proposed constructs is defined in a way that generalizes ordinary inheritance and allows a uniform treatment of type correspondences, partial inheritance and ordinary inheritance altogether. We argue that this semantics also leads to a type system with a richer semantic structure, a fact that manifests itself in higher potentials for code reuse. This structure reflects the conceptual organization of the problem domain, and supports the process of natural *software evolution* (Sect. 4.1).

Type correspondences and partial inheritance increase polymorphism, because they allow a given piece of code to be interpreted in new ways - in *metaphoric* ways. Following the terminology of *subtype* and *parametric* polymorphism employed in the context of inheritance and genericity respectively, we term the phenomena resulting from the inclusion of these constructs in programming *metaphoric polymorphism*.

[3] We assume that *power* receives a *positive* integer as argument.

As can be seen by the *INTEGER-MATRIX* example, metaphoric polymorphism raises some fundamental issues regarding type correctness. To start with, functions (methods of some type or free-standing) are supposed to accept arguments which are not subtype-compatible with the declared signature. These issues are briefly discussed in Sect. 4.2, and a full analysis is deferred to another paper.

Section 2 introduces type correspondences (2.1), and defines their semantics (2.2). Section 3 introduces partial inheritance (3.1), and outlines a real-life scenario that we believe testifies to the scaling up potential of the proposed constructs (3.2). Section 4 shows how these constructs support natural software evolution and structure (4.1), and discusses the new kind of polymorphism suggested here along with its implications on type correctness (4.2). Section 5 compares our approach to previous work. Section 6 concludes with suggestions for further research.

2 Type Correspondences

In this section we introduce the construct of *type correspondence* that enables programmers to take advantage of similarities between *existing types*. It establishes a partial mapping between two types, allowing appropriate pieces of code, originally expecting one, to work with the other. This construct is defined in 2.1, where it is informally described through some examples. The title *metaphoric polymorphism* is also explained in 2.1. A semantics treating type correspondences and inheritance uniformly is given in 2.2.

2.1 Construct Definition

A *type correspondence* is a programming construct that establishes a *partial* mapping between the attributes and methods of one type, called the *source* of the correspondence, to those of another, called the *target*. It may also specify some of the source's methods as being applied to the target. This will have the effect of legalizing calls, invoking methods listed in the **apply** part (along with their original implementations) on objects of the target's type, although that type does not have these methods listed in its definition, nor did it inherit them from an ancestor.

While executing the implementations for these applied methods on objects of the target type, references to attributes and methods of the source type will be interpreted in the target type according to the **map** part of the correspondence. In general, the mapping defined in the correspondence will be used whenever there is a need to interpret (on run time) a reference *exp.f* to a method or attribute, where the *intended* type of *exp* is the source type and the object at hand is of the target type. Such interpretations will very likely be needed while executing the applied methods, but they are also likely to be needed while executing any routine for which an object of the target type has been supplied while one of the source type was expected. This will be exemplified in a moment.

```
relate source_type to target_type
    [map source_att_or_method to target_att_or_method
         source_att_or_method to target_att_or_method
         .
         .
         . ]
    [apply source_method [as name_in_target ]
           source_method [as name_in_target ]
           .
           .
           . ]
end;
```

Fig. 1. A proposed syntax for the construct of type correspondence

Figure 1 shows a possible syntax for a type correspondence. As an example recall the case from the introduction, involving a class *INTEGER* with methods *multiply* (n:*INTEGER*): *INTEGER* and *power* (n:*INTEGER*): *INTEGER*, and another one *MATRIX* with a method *matrix_multiply* (some_matrix: *MATRIX*): *MATRIX*. Here is the type correspondence that allows the application of *power* to *MATRIX*, as discussed in Sect. 1:

relate *INTEGER* **to** *MATRIX*
  map *multiply* **to** *matrix_multiply*
  apply *power* **as** *matrix_power*
end;

Given this type correspondence, and assuming that *power* of *MATRIX* is implemented using *multiply*, the following call will be valid, where matrix1 and matrix2 are of type *MATRIX*[4]:

matrix1 := matrix2.*matrix_power*(5);

When the statement in the example arrives at execution, matrix2 is bound to some object of type *MATRIX*. Now, this type does not originally have a method called *matrix_power*, but in the type correspondence, *power* of *INTEGER* is applied to *MATRIX* as *matrix_power*. Therefore, the code for *power* will be executed. While executing, calls to *multiply* will be encountered, which will be interpreted as calls to *matrix_multiply*, because of the mapping of *multiply* to *matrix_multiply* in the type correspondence. We have thus added a new working method to the class *MATRIX* without having to implement it. In other words, the code for *power* of *INTEGER* has been reused in a new context, which is not a class inheriting from *INTEGER*.

[4] We assume that *power* receives a *positive* integer as argument.

The **apply** and **map** parts of a type correspondence have different roles. In fact, they could be separated to yield two different constructs. The mapping is the more essential part, enabling the use of certain pieces of code (that of *power* in this example) with types not originally intended for. *power* actually has two arguments: **self** and n, both *intended* to be integers. The type correspondence makes the text of *power* meaningful when the first argument (**self**) is a matrix and the second still an integer. The **apply** part makes *matrix_power* available as a new method of the type *MATRIX*, while establishing the code of *INTEGER*'s *power* as the one to be executed when it is called. Applying a method also maps it implicitly to the newly added one in the target. Thus, our type correspondence maps *power* of *INTEGER* to *matrix_multiply* of *MATRIX* although it is not explicitly specified.

The mapping established by a type correspondence in the **map** part may be useful regardless of the methods it applies, if any. For example, if *power* were a free-standing function expecting two integer arguments n and k, and returning n^k, then our type correspondence would enable calling it with a matrix as first argument, with no need for an **apply** part. As another example, consider a free standing-function *three_multiply* (n,m,k: *INTEGER*): *INTEGER*, which returns n×m×k. In the presence of the type correspondence relating *INTEGER* to *MATRIX*, *three_multiply* may be invoked with matrices as arguments:

matrix1 := *three_multiply* (matrix2, matrix3,matrix4);

three_multiply could of course be a method of some unrelated type, not just a free-standing function.

Note that the mapping is partial: *INTEGER* may have other attributes and methods which are not mapped to *MATRIX*, e.g. a method *prime:BOOLEAN*. Also note that in order for this to work, all that is required is for *power* to be implemented using *multiply*, but not in any specific way. For example, n^m may be computed by multiplying n $m-1$ times by itself, or it may proceed by computing n^2, then multiplying it by itself to yield n^4, and so on in $O(log(m))$ steps. Thus, type correspondences somewhat trade the principle of encapsulation for code reuse, but only to the extent, that in order to apply a method, one must know what other methods and attributes it refers to, not (exactly) how it works[5].

Back to the discussion of type correspondences in general, mapping a source's attribute or method f to h in the **map** part means that references to f directed at (= sent to) objects whose intended type is the source, but whose actual type is the target, will be interpreted as references to h. We shall have more to say on *intended types* in 2.2 .

In the **apply** part, it is possible, but not necessary, to give the applicable source's methods new names in the target's context. This is done using the **as** keyword. If f appears in the **apply** part of some type correspondence with target T, we say that f *is applied to* T (by that type correspondence). This means that it is possible to invoke f on objects of type T, even though f is not listed nor

[5] What exactly needs to be known and how to express it is an issue for further research. See Sect. 6.

inherited in T's definition. If f is applied to T without renaming, or some g is applied to T and is renamed as f, we say that f is *added* to T (by that type correspondence).

Here are some points and restrictions concerning the definition of type correspondences:

- Each source_attr_or_method is either an attribute (instance variable) of the source, a method of the source, or a method added to the source by some type correspondence (a method or attribute is *of* a type if it is defined there directly or it is inherited from some ancestor).
- Each target_attr_or_method is either an attribute of the target, a method of the target, or a method added to the target by some type correspondence.
- Attributes are mapped to attributes and methods are mapped to methods[6]. If method f_1 is mapped to method f_2 then f_1 and f_2 must have declared signatures of the same arity, that is they must have the same number of arguments, and either both are functions or both are procedures. There are no restrictions regarding relationships between argument types.
- Only methods may be applied, and all those that are applied must be of the source, or added to it by some type correspondence.

Clearly, it is not the case that a matrix argument to a routine may be supplied whenever an integer is required, as it is not the case that any method of *INTEGER* may be successfully applied to *MATRIX*. For instance, the method *prime:BOOLEAN* of *INTEGER* could not be applied to *MATRIX* because integer division has no counterpart in matrices (in the specific type correspondence of the example and also in principle). Similarly, a call matrix1.*matrix_power* (matrix2) would be illegal because there is no counterpart to the successor function in matrices. These limitations stem from the fact that type correspondences are *partial* mappings between types. The question of what is valid and what is not is an issue of type correctness, and is briefly discussed in Sect. 4.2. We note here that type correctness in the presence of type correspondences can be well defined and algorithmically verified. These results will be reported elsewhere.

The essential contribution made by type correspondences is the ability to *interpret* a given piece of code in new ways. The mechanisms of inheritance and genericity also give rise to a similar phenomena. With respect to these mechanisms, this ability is referred to as *subtype polymorphism* (for inheritance) and *parametric polymorphism* (for genericity). Following this terminology, we term the phenomena resulting from the inclusion of type correspondences (as well as *partial inheritance* introduced in Sect. 3) in programs *metaphoric polymorphism*.

The *INTEGER-MATRIX* example consisted of a single type correspondence. However, in the general case, a set of type correspondences may be combined for the purpose of a single reuse task. This is the case when the *pattern* to be

[6] In languages that also treat attributes as functions without arguments, such as EIFFEL, methods may be mapped to attributes (but not vice versa).

reused consists not only of a single class, but of a set of collaborating classes. For instance, if one class is part of another one, then partially relating the containing class could also require relating the contained one.

The following example illustrates this on a toy case. A more realistic example, that also includes partial inheritance, is given in 3.2. Consider a type *INTEGER_LIST* having a method *first*, which positions the cursor (some sort of pointer) on the first element of the list, a method *next:INTEGER*, that returns the element at the cursor's position and advances it one further, and a method *min_list:INTEGER*, which returns the minimal element of the list. Suppose also that we have a type *COURSE* with a method *first_student* that sets some pointer to the first student in the course according to the alphabetical order, and a method *next_student:STUDENT* that returns the student at the current position and advances to the next according to the alphabetical order.

We now wish to apply the *min_list* method of *INTEGER_LIST* to *COURSE* in order to find the student with the minimal grade. To do that, it is necessary to get down to the building blocks of *INTEGER_LIST* and of *COURSE*, that is to the level of integers and students. Suppose further that the type *INTEGER* has a method *less_than(INTEGER):BOOLEAN*, and that the type *STUDENT* has a method *has_lower_grade(STUDENT):BOOLEAN*. Now, assuming that the method *min_list* is implemented using the methods *first*, *next* and *less_than*, the following type correspondence will allow the desired application:

relate *INTEGER_LIST* **to** *COURSE*
    map *first* **to** *first_student*
        next **to** *next_student*
    apply *min_list* **as** *worst_student:STUDENT*
end;
relate *INTEGER* **to** *STUDENT*
    map *less_than* **to** *has_lower_grade*
end;

The observation that a complex structure, consisting of several type correspondences, can be used to relate two sets of collaborating classes in favor of some reuse task suggests that *frameworks* may be used in new ways when type correspondences are allowed for. This point becomes more prominent when *partial inheritance*, presented in Sect. 3, is brought into the scene.

Type correspondences may also be useful in combination with with *constrained genericity* as implemented in EIFFEL ([8]) and with *class substitutions* ([12], [11]). We cannot elaborate on that due to lack of space.

2.2 Semantics: Dispatch in the Presence of Type Correspondences

As discussed above, type correspondences allow functions (methods or free-standing) to be invoked with arguments different from the intended ones. This

means that when a call $exp.f(exp_1, ..., exp_n)$ invoking the method f on exp arrives at execution, it is no longer guaranteed that exp is bound to an object of type S, where S is the *intended type* of exp. Intuitively, the intended type of an expression is the type of exp as anticipated by the programmer who wrote it. If exp is a local variable, an attribute, or a formal argument, then its intended type is its declared type. If it is, however, a compound expression involving function invocations, then determining the intended type is a more subtle task. For example, the intended type of the expression *three_multiply*(n,m,k) is *INTEGER*, but the intended type of *three_multiply*(matrix1,matrix2,matrix3) is *MATRIX*. The latter is true despite the fact that *three_multiply*'s signature declares the resultant type as *INTEGER*. We do not give a precise definition of the intended type of a general expression in this paper but only state that this notion can be well defined. For the current presentation, an intuitive understanding will suffice. Alternatively, the reader may think of exp as being a simple expression (i.e. one identifier), in which case its intended type is its declared type.

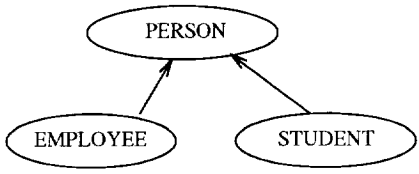
Continuing the discussion, the problem is that f is (supposedly) a method of exp's intended type, not of o's type, o being the object actually attached to exp when $exp.f(exp_1, ..., exp_n)$ arrives at execution[7]. The handling of this situation proceeds in two steps (T denotes o's type):

1. Identify which method of T (if any) is referred to by f. In the *INTEGER-MATRIX* example, when *power*'s text is executed on a matrix, a call n.*multiply*(m) might be encountered, where n is intended to be an integer but is in fact a matrix. Given the mapping defined by the type correspondence, *matrix_multiply* is identified as *MATRIX*'s method referred to by *multiply*.
2. Determine the implementation, i.e. the actual code, to be invoked. With *matrix_multiply* there is nothing special here: just use its implementation as defined in the class *MATRIX*. However, type correspondences contribute to this step too, when it comes to finding an implementation of an applied method. For example, when a call matrix.*matrix_power*(n) is encountered, it is the implementation of *INTEGER*'s *power* that should be invoked.

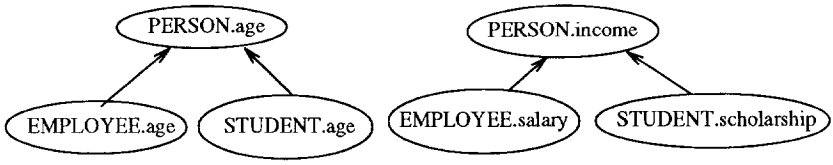
Before defining how these steps are carried out, let us point out that they conceptually exist even without type correspondences. To see that, consider the inheritance structure shown in Fig. 2, which assumes a language that supports attribute and method *renaming*, such as EIFFEL. Part (a) of that figure shows a directed graph whose nodes are types, representing an ordinary inheritance hierarchy between *PERSON*, *EMPLOYEE* and *STUDENT*. In part (b) the nodes are elements of the form $S.f$ where S is a type and f is a method or an attribute of S. There is an edge $T.h \rightarrow S.f$ iff T (directly) inherits S and $h = f$ or f is renamed h in T. Thus, for example, *STUDENT* inherits *PERSON* while renaming *income* as *scholarship* and leaving *age* as is. Now, if a call person.*income*() arrives at execution[8], where person is a variable of type *PERSON* bound to an

[7] Moreover, even if o's type has a method named f, it is not a-priori clear that it is this method that we want to invoke.

[8] We assume *income*, *salary* and *scholarship* to be functions with no arguments.



(a) An ordinary (type level) is-a graph



(b) An attribute-method level graph (G)

Fig. 2. Bringing the inheritance graph to the attribute-method level

object of type *STUDENT* (which is a subtype of *PERSON*), then *scholarship* is identified as *STUDENT*'s method referred to by *income*. This is because there is a path in the graph from *STUDENT.scholarship* to *PERSON.income*, *PERSON* being the intended type and *STUDENT* being the actual type.

The same attribute-method level graph may also serve for the second step - that of finding the implementation. This is done by conducting a search starting at the identified method (*STUDENT.scholarship* in the example) looking for a closet implementation[9].

We now extend this analysis to handle type correspondences as well. For the first step, i.e. that of identifying the actual type's method referred to, construct a graph $G_{identify}$ whose nodes are elements of the form $S.f$ where S is a type and f is a method or an attribute of S or a method added to S by some type correspondence[10]. Add an edge $T.h \rightarrow S.f$ to $G_{identify}$ when either of the following holds:

1. T inherits S in its definition and either f is not renamed and $f = h$, or f is renamed as h.
2. There is a type correspondence with source S and target T, in which f is mapped to h (in the **map** part).
3. There is a type correspondence with source S and target T, in which f is applied as h.
4. There is a type correspondence with source S and target T, in which f is applied without renaming and $f = h$.

[9] In the presence of multiple inheritance, either all inherited methods and attributes should have different names, perhaps after renaming (as in EIFFEL), or some linearization strategy should be employed during the search.

[10] Note that if T inherits S and f was added to S by some type correspondence then f will also be a method of T, and there will be a node $T.f$ in $G_{identify}$.

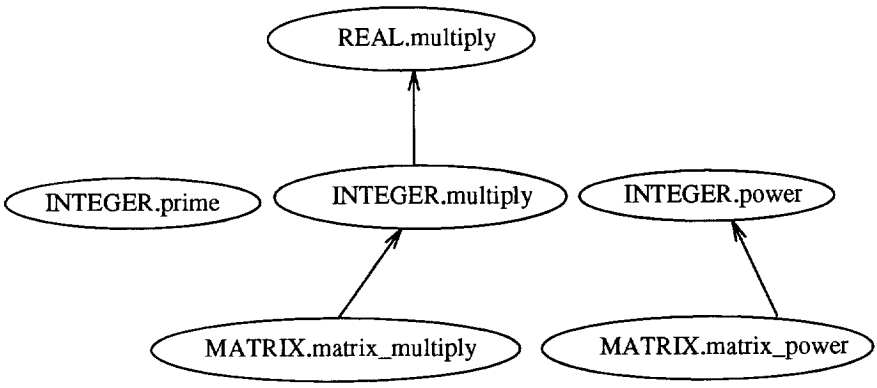


Fig. 3. $G_{identify}$ for the *INTEGER-MATRIX* example

The last two items reflect the fact mentioned in 2.1, that applying implicitly implies mapping. Figure 3 shows the graph for the *INTEGER-MATRIX* example. In that figure, one node for the type *REAL*, from which *INTEGER* inherits, is also included in order to show that type correspondences join the inheritance relation to yield one graph.

Given $G_{identify}$, we say that $T.h$ *corresponds* to $S.f$ if there is a path (of length 0 or more) from $T.h$ to $S.f$. Thus, the graph defines a reflexive binary *correspondence relation* between attributes and methods of types. $G_{identify}$ is required to be *unambiguous*, that is for any types S and T and any method or attribute f of S there should be at most one h such that $T.h$ corresponds to $S.f$. Ambiguity is also a potential problem without type correspondences. For example, in Eiffel such ambiguities must be resolved using the SELECT keyword.

For the second step, i.e. that of finding an implementation, we need a slightly different graph when type correspondences are in effect. Let G_{impl} be a graph similar to $G_{identify}$, only without the edges added due to **map** parts, that is without item 2 in the definition of $G_{identify}$.

With these graphs at hand, dispatching in the presence of type correspondences proceeds as in Fig. 4.

The correspondence relation between methods and attributes of types represented by $G_{identify}$ generalizes the ordinary inheritance relation between types. It contains all information present in the latter plus what is given by type correspondences. Given $G_{identify}$, we no longer care whether this or that edge is there because of inheritance or because of some type correspondence. Thus, the correspondence relation abstracts over the reasons that justified its construction. It directly represents a type system possessing a certain structure, and it can express a richer semantic structure than what can be expressed with an ordinary type-level inheritance relation. This richer structure results in higher potentials for code reuse.

Note, however, that the same *amount of code sharing* allowed for by a given type correspondence could be simulated using inheritance. For instance, in the

Let $exp.f(exp_1, ..., exp_n)$ be a statement invoking the method f on exp, where the intended type of exp is S. Suppose that when this statement arrives at execution, exp is bound to an object o of type T (because there are type correspondences, T is not necessarily a subtype of S). Let $G_{identify}$ and G_{impl} be obtained from the inheritance relation and the type correspondences altogether as described. To find the code to be executed proceed as follows:

1. *Identify which method of T is referred to by f*: search for a method h such that $T.h$ corresponds to $S.f$, that is, there is a path (of length 0 or more) in $G_{identify}$ from $T.h$ to $S.f$ (because $G_{identify}$ is unambiguous, there can be at most one such h). If the search is unsuccessful, terminate and return a "no such method" run time error (an appropriate type checking strategy should aim at detecting this at compile time). Otherwise, proceed to the next step.
2. *Search for an implementation*: perform a search on G_{impl} starting at $T.h$, looking for a closest node (to $T.h$) containing an implementation.

Fig. 4. Method dispatch in the presence of type correspondences.

INTEGER - MATRIX example, one could introduce an abstract class *MULTIPLIABLE* having methods *multiply* and *power*, the first of which is virtual (deferred). Then, letting *INTEGER* and *MATRIX* inherit from *MULTIPLIABLE*, each implementing *multiply* appropriately, will have achieved the same amount of code sharing. The main reasons why this observation does not render type correspondences superfluous are that they support the process of natural software evolution, and that they spare unnatural abstractions. See Sect. 4.1 for a full discussion.

Some object-oriented experts may object to the late addition of methods to classes, resulting from the **apply** part of type correspondences. The potential criticism is that in OOP all methods of a class should be defined in one place. However, what the **apply** does is no different from what ordinary inheritance does. With inheritance too, a class possesses methods not visible in its declaration. To overcome this, the environment may provide a *class flattener* ([8]), that gives a description of a class along with all its inherited attributes and methods. Such a tool may, and in fact should, bring in methods applied through type correspondences as well. This is also in line with our semantics that treats both constructs uniformly.

Finally, there are other statements and expressions besides those of the form $exp.f(exp_1, \ldots, exp_n)$ that need be analyzed when type correspondences are present. Among them are references to attributes and unqualified method calls. We cannot treat them here due to lack of space, but they do not introduce more difficulties than those discussed.

3 Partial Inheritance

A type correspondence partially relates types that exist independently of it, and in fact might have been defined long before the correspondence was identified

and recognized as beneficial. However, after realizing that the main point is the association of attributes and methods, rather than full types, to one another, it is only natural to provide a mechanism that derives one class from another while only taking *part* of its methods and attributes.

We propose *partial inheritance* for this purpose. There is nothing conceptually new here with respect to type correspondences: partial inheritance also gives rise to metaphoric polymorphism, that is the ability to metaphorically interpret a given piece of code. The only difference is that it establishes the (partial) correspondence while actually creating one of the classes. The new derived class is not (necessarily) a subtype of the one from which it is derived. Partial inheritance suggests itself as a natural generalization of (full) inheritance, and its semantics is also merged into the framework introduced in the previous section.

Suppression of attributes and methods in inheritance has been proposed before, and there are also experimental languages that support it. But in these languages, this is only conceived as a technical feature, not as a way of establishing a partial relation between types. It is in order to emphasize this point that we have chosen to present type correspondences first.

Combining partial inheritance and type correspondences provides a manner of reusing sets of collaborating classes in new ways. After defining the construct of partial inheritance in 3.1 below, we lay out in 3.2 an example that illustrates how a large software consisting of classes working together can be significantly expanded on the basis of reuse. If such an expansion were carried out without partial inheritance and type correspondences, then either the class hierarchy would need to be reorganized, or the new classes would need to be written directly without reusing existing components. Although that example regards a specific software, it strongly suggests that whole patterns could be reused from frameworks in new and unanticipated ways when constructs giving rise to metaphoric polymorphism are allowed for.

3.1 Construct Definition and Semantics

Figure 5 specifies an EIFFEL-like syntax for partial inheritance. Here is an explanation of the clauses mentioned in the figure:

- The **take** clause specifies those attributes of the inherited class that will become part of the inheriting class. The keyword **all** may be used to specify full inheritance.
- In the **rename** clause, some of the attributes and methods specified in the **take** clause may be given new names in the inheriting class.
- The **redefine** clause lists all methods whose implementations are redefined in the inheriting class. The methods listed here must appear among those defined in the inheriting class for the first time, where their new implementations are given.

```
class inheriting_class_name inherit inherited_class_name
      take
            (att_or_method,att_or_method,...,att_or_method) | all
      rename
            att_or_method as new_name
            att_or_method as new_name
            .
            .
            .
      redefine
            att_or_method,att_or_method,...,att_or_method

            Rest of the class definition

end class_name;
```

Fig. 5. A proposed syntax for partial inheritance

As a simple example, consider again the types *INTEGER* and *MATRIX*, but this time suppose that only *INTEGER* existed in the environment and we wished to derive *MATRIX*, including the method *matrix_power*, from it. Figure 6 shows how this is done using partial inheritance. Note that by this derivation *MATRIX* does not become a subtype of *INTEGER*. The latter has other methods such as *prime*.

```
class MATRIX inherit INTEGER
      take
            multiply, power
      rename
            multiply as matrix_multiply
            power as matrix_power
      redefine matrix_multiply

            New implementation for matrix_multiply

            Rest of the class definition

end MATRIX;
```

Fig. 6. Using partial inheritance to derive *MATRIX* from *INTEGER*

The semantics of partial inheritance is naturally described within the framework of Sect. 2.2: whenever C_1 partially inherits C_2, extend both $G_{identify}$ and G_{impl} as follows:

- Add nodes $C_2.f$, where f is an attribute or method taken from C_1 (if f is renamed as h, then add $C_2.h$ instead).
- Add edges $C_2.h \rightarrow C_1.f$ whenever f is taken and renamed as h, or it is taken without renaming and $f = h$.
- Add nodes $C_2.f$ for every new attribute or method f first introduced in C_2.

Having extended the graphs as described, dispatching in the presence of partial inheritance (and type correspondences, and full inheritance) proceeds as in Fig. 4 of Sect. 2.2.

3.2 An Example

The real-life scenario of this subsection involves partial inheritance and type correspondences that join in favor of a single reuse task. A real software would surely include many more classes, and those listed here are understood as forming only a partial picture. However, it should be clear from the example that a real software of this kind could be expanded in the same spirit.

Consider an airline company which has been in the air travel business for some time, during which a large object oriented software supporting its activities has been constructed for it. Among the classes of this software, there is one called *FLIGHT* that has attributes like *number, date, origin, destination, departure_time, arrival_time, passengers, carrier* and more. *carrier* holds an object of type *AIRCRAFT*, the one carrying out the flight. *passengers* holds a list of type *AIR_PASSENGER*.

The class *FLIGHT* also has several methods, one of which is *register_seats* (passenger_list: *LIST[AIR_PASSENGER]*, class: *AIR_CLASS*[11], smokers_flag : *BOOLEAN*) . This method operates by first invoking a method *get_seats* on the flight's carrier, and then distributing the seats returned by the aircraft among the passengers listed in *passenger_list*. The distribution proceeds by invoking the method *set_seat*(seat: *AIR_SIT*) on each passenger of the list. The result of the method *register_seats* will thus be reflected in the attribute *passengers* of the flight object (i.e. some of them, namely those included in *passenger_list*, will have seats). *get_seats*, a method of the type *AIRCRAFT*, has the following signature: *get_seats*(number : *INTEGER*, class: *AIR_CLASS*, smokers_flag: *BOOLEAN*): *LIST(AIR_SIT)*. It returns a list of available seats according to the desired number, class, and smokers/non-smokers preference.

Suppose that at a certain point, this company decides to expand its business to include railway traffic as well. It decides to treat train journeys much the same as flights. Naturally, the supporting software must be modified to handle this expansion. It would be of great efficiency if existing code could be reused without change and without harming what already works. We show how to use partial

[11] Do not confuse this with classes of OOP. Here we mean first class, business class and tourist class.

```
class TRAIN_JOURNEY inherit FLIGHT
      take
            number, date, origin, destination, departure_time, arrival_time,
            passengers, carrier, register_seats

            Rest of the class definition

end TRAIN_JOURNEY;

relate AIRCRAFT to TRAIN
    map get_seats to get_seats
end;

relate AIR_PASSENGER to TRAIN_PASSENGER
    map passenger_name to passenger_name
    map sitting_place to sitting_place
    map set_seat to set_seat
end;
```

Fig. 7. Deriving *TRAIN_JOURNEY* from *FLIGHT*

inheritance to derive a new class *TRAIN_JOURNEY* from the class *FLIGHT* with the help of some type correspondences.

Suppose a class *TRAIN* has been somehow defined (it could be derived from *AIRCRAFT* through partial inheritance, but we shall assume it was not). *TRAIN* also has a method *get_seats*(number: *INTEGER*, class: *TRAIN_CLASS*, smokers_flag: *BOOLEAN*): *LIST(TRAIN_SIT)* with signature and semantics similar to that of *AIRCRAFT*. Note, however, that instead of *AIR_CLASS* we here have *TRAIN_CLASS*. Also, the way *get_seats* operate in each type is different. For example, trains have wagons, and so looking for available seats is different. For that reason, a list of *TRAIN_SIT* is returned rather than a list of *AIR_SIT*. We also assume that these new types (*TRAIN_CLASS* and *TRAIN_SIT*) have been defined separately. Finally, we assume that a class *TRAIN_PASSENGER* have been defined, differing from *AIR_PASSENGER* only in keeping train seats instead of air seats, and, subsequently, in having the method *set_seat*(seat: *TRAIN_SIT*) with this different signature[12].

Figure 7 shows the derivation of *TRAIN_JOURNEY* from *FLIGHT* using partial inheritance and the type correspondences needed for it. Note that quite a number of methods and attributes have been inherited by *TRAIN_JOURNEY*, and there could probably be many more if the example were complete. Also note that there may be other attributes and methods of *FLIGHT* not inherited by

[12] *TRAIN_PASSENGER* could certainly be derived from *AIR_PASSENGER* through generalized inheritance.

TRAIN_JOURNEY such as *maximum_weight_allowed*, *free_meal_menu* etc. This is no wonder since *TRAIN_JOURNEY* is not a subtype of *FLIGHT*.

4 Discussion of Metaphoric Polymorphism

4.1 Support for Natural Software Evolution and Structure

In Sect. 2.2, we have remarked that the same amount of code sharing achieved by type correspondences can be simulated via (full) inheritance. To simulate a type correspondence relating T to S, define a new class encapsulating those attributes and methods of T which are mapped or applied to S, and let T and S inherit from this new class, redefining implementations as needed. The same technique may be applied to simulate partial inheritance via full inheritance. However, this is only a technical simulation that in many cases does not fit well into the *development process*. Consider the following points:

- If one or both of the two types related via a type correspondence existed before the correspondence was identified and recognized as beneficial, then this simulation requires to *reorganize* the class hierarchy, so that the existing type(s) will inherit from the new abstract class. This means touching a working code - a fine potential for problems. This argument applies equally to partial inheritance, in case the inherited class already existed as a functioning component. Even if the language supports a *generalization* mechanism as proposed in [13], then using our proposed constructs will still be more appropriate in many cases: see the next two points, the rest of this subsection and the comparison to related work in Sect. 5.
- Partial inheritance along with type correspondences may be the only way to adapt *supplied* components or frameworks, as it is unreasonable to let imported components inherit user defined classes.
- The new abstract class might turn out to be an ad-hoc class, not representing any natural abstraction. One should aim at minimizing the number of such classes present in the system.

Let us back-up these arguments by a deeper rationale. As we have been emphasizing, the proposed constructs are motivated from metaphors. In many cases, metaphors of thought precede abstractions, and in that sense they are more basic. For example, it is unlikely that the notion of *vehicle* was conceived just after the invention of the first vehicle. Rather, there probably was some sort of vehicle, say a carriage. Then another sort emerged, say a train, and terms from one were applied to the other. For example, the notions *driver* and *passenger* were applied from carriages to trains. Then perhaps a new sort of vehicle came about, and some of these notions were applied to it as well. After some time, enough concepts shared enough terms to *justify* the encapsulation of these common terms into the new abstraction of *vehicle*. From that time on, carriages, trains, cars etc. were considered vehicles.

Now, the key word is *justify*: not all metaphors end up as abstractions, and if they do, it is after some time. One might say that there is a trade off in the conceptual network: we can have fewer concepts related by metaphors, or we can pay the price of reorganizing the network, ending up with simple taxonomic (sub-concept) relationships, but with more concepts.

In view of this discussion, we propose the following principle argumentation for the inclusion of type correspondences and partial inheritance in object oriented languages:

1. **The underlying rationale.** Metaphors that partially relate concepts are first class citizens of language and thought. It therefore makes sense to incorporate appropriate counterparts in programming to yield a more elaborate organization of the type system, reflecting conceptual structure, and resulting in higher potentials for code reuse.
2. **Support for natural software evolution and structure.** The conceptual network often evolves through metaphors. Partial inheritance and type correspondences can be used to reflect this *evolution* in software. This means that functionality can be added to software in a manner that parallels its discovery. In practice, it means that components may be reused, without change and without adding any unnecessary abstractions, in ways unanticipated by their developers. Abstraction can take place at a later stage, if and when it is justified. In this development paradigm, software evolves along with its designers' way of thinking. As is the case in the conceptual level, the question of justification involves a trade off: either leave things as they are, or pay the price of reorganization to gain a simpler structure, but with more types.
3. **Adaption of supplied components.** Since frameworks or supplied classes cannot be made to inherit user defined classes, partial inheritance and type correspondences may be the only way to adapt them to a given application. As noted in Sect. 3, we conjecture that these constructs open a new range of possibilities for the exploitation of frameworks.

Finally, we note that the process by which a type correspondence or a partial inheritance clause is replaced by a new abstraction can be automated. This suggests once again, that abstracting should be delayed until it is justified, whence it can be done automatically, requiring the user only to supply a meaningful name - a task that can be carried out properly just when the abstraction is indeed meaningful.

4.2 Polymorphism Based on Semantic Correspondences

Most typed object-oriented languages support subtype polymorphism and dynamic method dispatch. This means that an identifier may be bound on run time to objects of any type which is a subtype of its declared type, and that a method invocation on that identifier will be dynamically dispatched according to the type of the object to which it is bound.

This sort of polymorphism is entirely based on the subtype relation between types. Indeed, the *definition* of what it means for one type to be a subtype of another was engineered to suit subtype polymorphism. Such a definition may be found in [1]. It implies that T_1 is a subtype of T_2 iff objects of type T_1 may appear wherever objects of type T_2 are expected. In [3] it was shown that inheritance and subtyping are distinct, and that identifying them may lead to insecure type systems. This is because inheritance does not imply subtyping, on which safe substitutability must be based. EIFFEL is singled out there (and previously in [4]) as possessing this problem.

Nevertheless, substitutability must be based on subtyping only in as much as one insists on a type checking strategy that never looks back on anything but a class' declared interface once it has been successfully compiled. Although such strategy is surely desirable for obvious reasons, one's adherence to it should be weighted against the benefit that might be gained from constructs that render it inappropriate. Of course, some other strategy should be suggested to handle such constructs[13].

Throughout this paper we have demonstrated the advantages of basing polymorphism on *semantic correspondences* between types, not (just) on subtyping. $MATRIX$ is not a subtype of $INTEGER$, but because the programmer semantically related them via a type correspondence, the former may be beneficially substituted for the latter in *appropriate contexts*. But to enjoy this flexibility, one must abandon the conception that T_1 is allowed to be substituted for T_2 only if it can be substituted for it in *any* context, that is only if T_1 is a subtype of T_2. It is enough that the substitution be valid in the *given* context, represented by the program or class library at hand. That is to say, a program is quit safe if it issues a call *three_multiply* (matrix1,matrix2,matrix3) but does not issue calls such as *three_multiply* (matrix1,n,m) or matrix1.*matrix_power*(matrix2).

The type safety of programs or class libraries allowing such polymorphism can be well defined and algorithmically verified through an appropriate type checking procedure. Such a procedure cannot exclusively rely on a class' declared interface once it has been compiled, but it does not necessarily have to involve analyzing a class' text more than once. Roughly speaking, the compiler can generate appropriate constraints while compiling a class, listing the conditions that must be met by arguments supplied to methods of that class. For a given method, these constraints may be thought of as a "generalized signature". Then, only this information, not the class' text, will be re-consulted on need.

Note that the declared signature of routines no longer restrict arguments to the specified types. Nevertheless, user type annotations do not loose their meaning: they serve to interpret routines by providing the *intended types* of identifiers and (indirectly) of compound expressions. A routine's text is interpreted with respect to these intended types given the correspondence relation and the actual

[13] The *system level validity* originally intended for EIFFEL ([8]) was a strategy meant to resolve its type checking problems, that resulted from the (justified) desire not to restrict substitutability to subtypes. Recently, however, Meyer declared his intention to adopt a new rule ([9]) which will actually restrict polymorphism to subtypes.

provided types. But still, when one looks at a given routine's text, one need not be bothered with the possibility of this text being interpreted metaphorically: for all he or she is concerned, the text is understood, maintained, assessed for quality etc. as if the types handled by the routine are those actually declared. Later on, someone may take this routine and apply it - or supply arguments - other than the ones for which it was originally written. Deciding whether or not such use will be valid is the job of the type checking procedure. Thus, user type annotations still keep their most important advantage of *readability*, because when one *reads* a routine, she or he need not concern themselves with its possible metaphoric interpretations.

Finally, we note that the definition of type correctness should preserve the *context independence* of subtype substitutability, that is it should imply the unconditional safety of providing arguments to a routine which are subtype compatible with its declared signature. This will ensure that no extra work will be done by the compiler for subtype polymorphism relative to the situation in existing languages. Results to that effect will be reported elsewhere.

5 Related Work

This section discusses some related work and compares it to ours.

The as-a relationship. In [10] Mitchell et al. propose a relationship between classes, called **as-a**, that is meant to support code reuse. They argue (as others did previously) that people do not always use inheritance in ways that reflect **is-a** relationships between heirs and ancestors, but they often tend to use it for reasons of convenience, in order to reuse some implementations. As they say, "...programmers do use inheritance to reuse code from one class into another, with no intention of substituting objects of one class for objects of the other via dynamic binding and with no intention of explaining or legitimating the use of inheritance in terms of an is-a relationship in the modeling domain".

The authors of [10] propose not to fight this tendency, but to support it in a way that will ensure safe usage of such implementation inheritance. To account for this, they propose to allow what we called here partial inheritance, and to support its correct usage by an **as-a** relationship. This relation actually involves three parameters: a class C_1 **as-a** C_2 **for** some operation o. The idea is that if this holds then o may be safely inherited from C_2 to C_1. **As-a** relationships are established in a formal system that uses equations relating attributes in both classes, to prove certain implications between the post and pre-conditions of the operation at hand.

There is an important fundamental difference between their approach and ours. The authors of [10] stress that the problem they address is that of implementation inheritance, that is the apparent desire of programmers to inherit without any conceptual basis. This is exemplified in the quotation given above. Consequently, they devise a tool that enables one to establish that it is indeed correct to inherit in such circumstances (when it is correct). But they are not

interested at all in the relationship that is permanently established via such (partial) inheritance between the two types[14].

Our approach, on the other hand, is entirely based on the observation that one should allow types to be partially related because it reflects *conceptual metaphors*. Partial inheritance is conceived as just one way of achieving this. Contrary to Mitchell et al. we consider such relationships to be a strong basis for substitutability, that is for semantic (or metaphoric) polymorphism. Indeed, that is the main point that we wish to convey (see Sect. 4.2). In our approach, a matrix **is-a**[15] integer as far as it concerns *appropriate contexts*. This approach also led us to (first) introduce the construct of type correspondence that may relate two existing types, while they talk only of inheritance. There are many other essential and technical differences between these two approaches that we could not discuss due to lack of space.

Generalization. In [13] Pedersen proposes a construct that generalizes over a set of classes to yield one parent class. The set of methods of the new class is the intersection of the sets of methods of all classes generalized over (minus some methods explicitly removed). This construct indeed offers a way of reusing an existing class from which, in our terms, we would like to *partially* inherit. This proceeds by first abstracting over the existing one while removing unnecessary methods, and then inheriting. While such a construct should be useful, there are cases in which it forces the inclusion of unnecessary abstractions in the class hierarchy, and others in which it is not applicable (see Sect. 4.1). We believe that this construct should co-exist with ours to provide maximum flexibility. As argued in 4.1, however, type correspondences and partial inheritance will usually better suit the natural evolution of software. Moreover, given these constructs, generalization at a later stage can be automated.

Fine-grain inheritance. This notion was proposed by Johnson and Rees in [5]. It is a strategy according to which classes should be as minimal as possible (feature-wise), so as to make them fully inheritable. They urge designers to define many small classes, representing "small concepts". Without starting a discussion on what makes up a concept, it seems inevitable that followers of such strategy will end up defining many ad-hoc, perhaps conceptually unclear, classes. We believe that the problem addressed by [5], that of reusing a library in ways not predicted by its authors, should find its solution through a paradigm that accounts for elaborate relationships between real and obviously justified concepts. We hope to have made a contribution in this direction.

Ada-style genericity. In Ada, it is possible to write generic packages that are parameterized not only by types, but also by operations on these types. It should therefore be possible to write one generic package that implements *power* while parameterizing on its first argument's type and on the multiplication operation. Apparently, there is no need to abstract over *INTEGER* and *MATRIX* for this package (i.e. there is no need to define a type *MULTIPLIABLE*), which is similar

[14] And maybe for that reason they speak of *classes*, not *types*.

[15] There is no mistake here - we did not mean **as-a**.

to our case. But the resemblance is only apparent, since the abstraction is there anyway: it is this generic package that constitutes the abstraction. Whoever writes it thinks of multipliables, that is (abstract) types that have a multiplication operation. Our approach genuinely avoids the abstraction. It makes it possible to use *power*, written exclusively for *INTEGER* and even as part of it, for matrices. In that sense, our approach offers an alternative to generic programming because there is *really* no need to abstract in order to reuse. Of course, genericity is very useful when the abstraction is justified and made on time.

6 Conclusion and Further Research

We have proposed the constructs of *type correspondence* and *partial inheritance* for object oriented programming, both of which are motivated by metaphors of natural language and thought. They have been shown to introduce a new kind polymorphism into programming, called *metaphoric polymorphism*, which results in previously unavailable code reuse potentials. The semantics of these constructs was defined as a generalization of ordinary inheritance, and was shown to yield a type system with a richer semantic structure. These constructs were shown to support natural software evolution and structure.

A first and most important issue for further research is type correctness in the presence of metaphoric polymorphism, as discussed in 4.2. We shall report results to that effect in future works.

Because **is-a** relationships play a central role in knowledge representation, and in *semantic nets* in particular, it may prove beneficial to examine the applications of the proposed semantics, involving a correspondence relation in a level lower than types (concepts), to this field.

Another issue is that of *method specification*. In order for one to *consider* applying a method in a type correspondence, one must know something about the methods and attributes referred by that method's implementation and how they are inter-related[16]. A similar knowledge is required in order to consider providing routine arguments which are not subtypes of the required ones. What that "something" is, and how it may be expressed, is a matter for further research. Algebraic specifications are a possible direction, but less formal methods could also be considered. Recently, Stata and Guttag ([16]) proposed to provide *specialization specifications* to programmers that adapt classes by inheritance, which are different from the specifications given to other programmers, who just use them as black box components. Their specialization specifications divide the methods of a class into independent groups of cooperating methods. Then, only entire groups can be overridden in inheritance. Their approach may prove beneficial for our case.

We have conjectured in this paper that metaphoric polymorphism opens new opportunities to use *frameworks*. Finding some large scale examples to that effect is one more research option.

[16] Whether or not the application is in fact correct will be determined by the type checking procedure.

While presenting the metaphor motivated constructs, we have advocated the stance that reflecting conceptual structures and mechanisms in programming is practically beneficial. We believe that many more contributions can be made to programming on these grounds.

Acknowledgments

We thank Yossi Gil and Ari Rappoport for their comments on earlier versions of this paper. We also thank the referees for their comments, especially the one who provided a long and detailed list of valuable suggestions.

References

1. R.M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4), 1993.
2. G. Booch. *Object-Oriented Analysis and Design with Applications.* The Benjamin/Cummings Publishing Company, Inc, 1994.
3. E.R. Cook, W.L Hill, and P.S. Canning. Inheritance is not subtyping. In C.A. Gunter and J.C Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming.* The MIT Press, 1994.
4. W.R. Cook. A proposal for making eiffel type-safe. *The Computer Journal*, 32(4), 1989.
5. P. Johnson and C. Rees. Reusability through fine-grain inheritance. *Software-Pratice and Experience*, 22(12), December 1992.
6. G. Lakoff. *Women, Fire and Dangerous Things: What Categories Reveal About the Mind.* The University of Chicago Press, 1987.
7. G. Lakoff and M. Johnson. *Metaphors We Live By.* The University of Chicago Press, 1980.
8. B. Meyer. *Eiffel, the Language.* Prentice Hall, 1992.
9. B. Meyer. Beware of polymorphic catcalls. Personal research note, http://www.eiffel.com/doc/manuals/technology/typing/cat.html, 1995.
10. R. Mitchell, J. Howse, and I. Maung. As-a: a relationship to support code reuse. *Journal of Object-Oriented Programming*, 8(4), July/August 1995.
11. J. Palsberg and M.I. Schwartzbach. *Object-Oriented Type Systems.* John Wiley & Sons, 1994.
12. J. Palsberg and M.I. Schwartzbach. Type substitution for object oriented programming. In *OOPSLA/ECOOP '90 conference proceedings, ACM SIGPLAN Notices, Volume 25, Number 10*, October 1990.
13. H. Pedersen. Extending ordinary inheritance schemes to include generalization. In *OOPSLA '89 conference proceedings, ACM SIGPLAN Notices*, 1989.
14. J. Rumbaugh. Dishinerited! examples of misuse of inheritance. *Journal of Object-Oriented Programming*, 5, February 1993.
15. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.
16. R. Stata and J. Guttag. Modular reasoning in th presence of subclassing. In *OOPSLA '95 conference proceedings, ACM SIGPLAN Notices, Volume 30, Number 10*, October 1995.