Environmental Acquisition
— A New Inheritance Mechanism*

JOSEPH GIL       DAVID H. LORENZ
The Faculty of Computer Science,
Technion—Israel Institute of Technology,
Technion City, Haifa 32000, ISRAEL;
Email: {yogi | david} @CS.Techinon.AC.IL

LPCR Report #9507
December 1995

Abstract

Nature vs. Nurture? The debate has obsessed the minds of psychologists and philosophers for many years. However, for the object-orientee, it has never been a problem: an object inherits all of its properties. In this work we ask if an object should not be subject to environmental effects. We answer this question in the affirmative by demonstrating many cases in which the character of an object must be affected by the environment it is put in. We present a new abstraction mechanism—Environmental Acquisition—which allows a component to inherit properties from its enclosing composite(s). The need for environmental acquisition is demonstrated in several application domains. It is shown that the absence of environmental acquisition may lead to the creation of cumbersome programs. We propose a strongly typed model for environmental acquisition that allows static type checking of programs exploiting this mechanism, and compare it to several other mechanisms including ordinary inheritance and delegation.

*Available through World Wide Web as [20].
1 Introduction

“Nature vs. Nurture?”¹ This long standing question has been occupying the minds of philosophers, psychologists, laymen and even physicists [45] for years. The dispute is over the relative importance of heredity and environment in determining the makeup of an organism. However, for the object-orienteer², it has rarely been a problem: The basic character of an object, sometimes called “behaviour” in the Object Oriented (OO) jargon, is determined at birth (instantiation), and not by the household (the composite object) of which it is a part. This simplistic sweeping claim is, as are all such claims, false for humans. How true is it for objects? How should the “nurturing” of an object affect its manners? Can such influence be dealt with in a (type) safe manner?

In this paper we address these questions. We explain what is and what is not “the influencing environment” of an object. We show that there are many important cases, both in the problem and program domains, in which the need for environmental effect naturally arises. Furthermore, we propose a new abstraction mechanism, environmental acquisition, and discuss its possible realization in a strongly-typed programming language, while giving a few examples from a concrete application domain.

This paper does not provide solutions to all the problems it raises. Rather, it presents a framework for addressing the issues involved in the many aspects of environmental effects.

1.1 Motivation

To give the reader a taste of the motivation for this research, consider the following example which may occur in an automobile industry application: An object of a class Car depicted in Figure 1, is a composite which comprises components such as objects of class Door.

![Diagram](https://via.placeholder.com/150)

Figure 1: Environmental acquisition in the problem space: Class Door acquires its colour from Sedans and Hatchbacks alike.

Suppose that it is known that a car is coloured red, then we are likely to infer that its doors

¹ Also known as the environmental-heredity controversy.
² or, should we write, object-orientalist?
are red as well. However, although the door “inherits” its colour from the car of which it is part, it would be wrong to derive Door from Car. Colour-inheritance is related to the “in-a” (reverse-“has-a”) link which binds doors to cars.

This is further exemplified by examining Car sub-classes, say Sedan and Hatchback, which are distinguished among other things by their number of doors. Class Door inherits its colour from Sedans and Hatchbacks alike, just as it would from another hypothetical class Airplane which does not stand in an “is-a” (“a-kind-of”) relationship with Car. We call this kind of inheritance Environmental Acquisition (acquisition for short) and distinguish it from Ordinary Inheritance (inheritance for short) in its common meaning in object oriented programming. The term inheritance by itself will hereafter refer to ordinary inheritance whereas acquisition will refer to environmental acquisition. Observe that since acquisition binds objects and not their classes, it does not (indeed it cannot) induce any sub-type relationship. This is in contrast to the common role of inheritance in OO Programming (OOP).

The above example is drawn from the problem space. Another example, which belongs in the program space, is that of class objects. In a pure object oriented programming model, such as that of Smalltalk [23], classes are also objects. The concept of class objects occurs even in less pure models such as Objective-C [12], and SOM [16]. A class object is an instance of a meta-class Class\(^3\) which provides a template for the instantiation of objects in the class and defines their behaviour. If a class \(\text{C}_2\) is derived from \(\text{C}_1\), then the class object of \(\text{C}_2\) has a “super” link to its containing class object of \(\text{C}_1\) as shown in Figure 2. Inheritance is realized by propagation of properties of \(\text{C}_1\) along that pointer from \(\text{C}_2\). This can be thought of as:

\[ \text{C}_1 \rightarrow \text{Class} \rightarrow \text{C}_2 \]

Figure 2: Acquisition in the program space: In Smalltalk the meta-class Class defines the propagation of properties along super links of class objects. Thus inheritance is implemented via acquisition in the meta-class Class.

\(^3\)For the purposes of the example, it is sufficient to assume that there is only one meta-class and that abstraction stops at Class. That is, there are no meta-meta-class, meta-meta-meta-class etc. However, the example becomes even more interesting, albeit more complex, if these are allowed.
of as acquisition in the meta-level. Part of the role of the meta-class \textit{Class} is to define and implement the process of this propagation.

The fact that the propagation of properties across links is not part of the usual object model contributes to the complexity of understanding and programming with class objects and meta-classes. Assuming a single-inheritance scheme, a view which mitigates this difficulty is that of class objects as representing sets. The class object \textit{Car} models the set of all cars. The class objects \textit{Sedan} and \textit{Hatchback} model their corresponding sets, which are \textit{part-of} the class object of \textit{Car}. Now, mechanisms which allow components to environmentally acquire attributes from their respective composites put what we called above “colour-inheritance” of doors from cars, along with the inheritance of \textit{Sedan} from \textit{Car}, at the same conceptual level.

1.2 Environmental acquisition in a nutshell

Generally, an OO system includes two hierarchies, \textit{class} inheritance and \textit{object} composition, as illustrated in Figure 3. Despite similarity lines drawn above, there are important differences between acquisition and inheritance which are highlighted by the distinction in classical (class-based) OO languages between shared properties and between particular properties of an object. \textit{Shared properties} are those which are determined by the object’s class. They include \textit{behaviour} (declaration and definition of methods) and \textit{structure} (declaration of instance variables). \textit{Particular properties} may be different in different objects of the same class. They include the object’s identity and \textit{state} (current values of instance variables). Inheritance pertains to classes and as such can be viewed as a means of abstraction over

![Figure 3: An OO system typically includes two hierarchies: The “child-of”-inheritance and “is-a”-aggregation hierarchies. The first is a hierarchy of classes, the second is a hierarchy of objects, and object are related to classes via “is-a” dashed links.](image-url)
shared properties. In contrast, acquisition can be viewed as a means of abstraction over particular properties.\(^4\)

Consider an object \(a_1\) of class \(C_1\) which has an instance variable \(a_2\) of class \(C_2\) as depicted in Figure 4. All shared attributes of \(a_2\) are determined by \(C_2\). Traditionally, the particular properties of \(a_2\) are independent of both the shared and the particular properties of \(a_1\).

![Figure 4: The environment of \(a_2\) consists of the particular properties of \(a_1\) as well as its \(C_1\) shared properties. This is in contrast to the traditional approach where particular properties of \(a_2\) are independent of both the shared and the particular properties of \(a_1\).](image)

It should be stressed that the acquisition we propose is strongly typed, and that specific ways are provided for the programmer of an object to infer about all of its potential environments and deal in a type safe manner with a concrete environment in which it exists. Moreover, a component can be dynamically move to a different composite, resembling perhaps dynamic inheritance.

Acquisition is also subject to polymorphism\(^5\). *Environmental Polymorphism* means that there are many possible variant behaviours of objects of a given class, and that the precise behaviour is dependent on the environmental effect. The terms acquisition and environmental polymorphism are complementary, just as inheritance and *subtype polymorphism* \(^7\).

\(^4\)One may argue that in the case of inheritance an object inherits particular properties from its sub-objects. With acquisition the “subobject” would had inherited from the object.

\(^5\)For a taxonomy of types of polymorphism see Strachey [42] and Cardelli and Wegner [7].
are. Environmental polymorphism is different from subtype polymorphism: With subtype polymorphism, the code applicable to a certain type (class), be it part of that type definition or not, is also applicable to all of its subtypes (sub-classes). The “behaviour” of the code is therefore parameterized by the actual object it operates on. With environmental polymorphism, the behaviour of an object is parameterized by its surrounding environment.

Acquisition is also different from programming with exemplars [30, 4] in the same way that classical class-based languages are different from languages such as SELF\(^6\) [47]. Although acquisition can be imitated (to a known extent) by the delegation mechanism, such an imitation does not match a disciplined use of this paradigm as enforced by built-in lingual support. This distinction is similar to that of high level languages with assembly. Although high level languages can be imitated in assembly and are therefore less powerful, the use of high level constructs promotes better programming practices. Furthermore, as we shall see, it is possible to practice type-safe programming with acquisition, a trait which the unharnessed power of delegation excludes.

1.3 Outline

In this paper we try to investigate and promote the concept of acquisition. The rest of the paper is organized as follows: In Section 2 we present several important application domains in which acquisition emerges naturally. We also show that lack of support for acquisition leads to some cumbersome implementations. Section 3 lists the requirements of a good acquisition system. In Section 4 it is explained why strong-typing should and can be preserved in such a system. Section 5 sheds some more light on acquisition by reviewing some of the basic terms and concepts of inheritance, such as abstract classes and overriding, and identifies their analogs in acquisition. In Section 6 the proposed acquisition is compared with other models, including dynamic inheritance, delegation, genericity, composite object support [26] and complex associations [28]. Section 7 gives the conclusions and raises several open questions.

2 Application Domains

Containment hierarchy is a ubiquitous concept in programming methodology in general, and specifically in the object oriented paradigm. Many of the object oriented analysis and design methods even devote a special notation to it [3, 9, 40].

In this section we show that there are abundant cases of containment hierarchies where

\(^6\)There is no need in a “single-hierarchy” system with only one kind of objects, as in SELF, for meta-classes because objects describe themselves. SELF provides, however, dynamic inheritance in the form of delegation.
contained objects have different behaviours depending on their surrounding environment. We concentrate on concrete program-space examples and give five application domains in which this phenomenon occurs naturally. For warm-up we also present the classical example of real-life knowledge representation:

**Clyde the elephant** Consider the well-studied elephant called Clyde [14]. His expected behavior does not depend only on his intrinsic character but also on that of its owner and on its surrounding environment. For example, put in a zoo environment, Clyde would presumably be quiet and passive. Clyde of the jungle would be employed for transportation, and would be missing one tooth. Clyde the circus elephant would be harmless, but somewhat gloomy.

### 2.1 GUI Systems

Object oriented Graphic User Interface (GUI) frameworks typically organize screen elements: windows, views, dialog boxes etc., in a view tree. Systems of this sort are for example, Turbo-Vision and its descendent OWL [46, 37], InterViews [31], and that of NeXT [44]. Responsibilities, such as screen drawing and handling input events, are distributed down this environmental-tree. A screen element to which responsibility is delegated from its environmental-parents, also environmentally acquires some of its parents’ traits. In Turbo Vision [46] for example, we see the following acquired traits:

**Origin of the coordinate system** The coordinate system of a screen element is relative to that of its parent.

**Error handling** The routine to call in case an unrecognized event occurs is that of the parent.

**Control Flow** Modal screen elements are sub-trees which, when activated, disable all elements external to them. Examples of modals are yes-no message boxes and the whole application itself. When a modal element terminates, it returns control to its nearest enclosing modal element. The nearest enclosing modal element is implemented as an environmentally acquired attribute which exists in all screen elements.

**Attributes Palette** Screen colours (aka attributes) of a screen element are given as indices of a palette table stored in one of the screen element ancestors. The values stored in this table serve in turn as indices to an ancestor of this ancestor and so on until the application global attributes are reached. This acquisition mechanism is designed for flexibility and power, but it makes programming and reasoning with palettes a complex task.
InterViews uses acquisition for computing constraints on an object’s size. Other GUI systems use acquisition for other purposes e.g., context sensitive help, where a screen element acquires its response to a help request from its enclosing element but may override it to support more specific help.

These GUI systems are written in languages such as C++, Objective-C, and Object-Pascal that do not support acquisition as an environmental inheritance mechanism. Acquisition must be therefore emulated, usually by a complicated web of pointers and schemes for call-back. Beyond unwieldy complexity, this results in difficult to understand features (as in the attribute palettes example), or in non-safe programming, (e.g., is it always guaranteed that an exiting modal will find an enclosing modal?).

2.2 Graphic modeling

The intricacies of Turbo Vision’s attribute palette are only multiplied in the realm of high resolution graphics modeling 3-D objects. Here it is necessary to address issues of acquisition of many more kinds of attributes: line styles, colour, shading, textures, modeling type, transformation matrix, etc., spread along very complex objects. The graphics community recognizes that the ensuing questions are difficult [15, Section 9.2]. However, there is at least one major graphic standard which includes acquisition [39].

2.3 Text processing

Emphasized text in LATEX [29] normally prints in italics. However, an emphasized within an emphasized block prints in roman (as demonstrated in this sentence). This is only one of the many examples in the domain of desktop publishing systems where the behaviour of text elements is strongly dependent on their surrounding environment. Systematic approaches to document processing, such as SGML [24, 6] and RTF (Rich Text Format), use a hierarchical representation, and let attributes such as type-face, text-size, and bounding boxes be environmentally-inherited by elements from their surrounding elements. Some modern word-processors (e.g., Dagesh [13]) even explicitly use the word “inheritance” to denote what we call acquisition.

The usage of acquisition is very evident in the scoping model of TEx [27]: the value of all macros, declarations, registers etc., is acquired at each point from the innermost enclosing scope in which they are defined. Acquisition and Environmental Polymorphism are part of the reason why programming TEx macros is so notoriously difficult. A macro is a polymorphic object whose behaviour depends on the values of the commands and on other macros that it calls at the time of activation. This is further complicated by the fact that a macro may change its own definition during its execution. Therefore, recursive calls may
amount to something totally different than our usual understanding of recursion.\footnote{Another complication arises from allowing macros to modify both the lexical and the syntactical aspects of the embedded programming language. This makes \TeX{}'s grammar highly noncontext free.}

### 2.4 User defaults in an operating environment

Today’s fancy GUI windowing environments (e.g., MS-Windows, Macintosh, NeXT, etc.) allow the attachment of various defaults and user preferences to files and other resources. For example, one may attach a word processing application, customized and configured appropriately, to a certain type of documents. On clicking on a document’s icon, the corresponding application will be invoked in order to edit it. If the user manipulates many different projects, or if a large multi-user environment is to be supported, there is a great need to carry out this attachment in an orderly manner.

Even in a simple Unix environment such a default system is needed: Different users have different preferences, which may further depend on the types of files and their location. Upon editing a file of an unknown type located anywhere under the Programs directory, the editor should be in the programming mode. If the same file is under a Documents directory, defaults should change accordingly. This behaviour may be different depending under which home directory these directories reside.

Acquisition, together with inheritance hierarchy of types of files and other resources, is the appropriate way of setting defaults. Its absence brings about a major source of confusion for naive users and a cause of headache for the system administrator. Current solutions in Unix use a tangled mixture of compile-time flags for applications, system-global, user and directory initialization files (\texttt{.Xrc} in the Unix jargon) together with environment variables settings. Shared (“networked”) installation of many applications in MS-Windows is next to impossible.

### 2.5 Language processors and reverse engineering

Another application domain which calls for acquisition is that of processing formal (programming) languages: compilers, interpreters, automatic generation of test cases, computing metrics, etc. If the language subject to processing belongs in the Algol family, or even more generally, if it has static binding, then acquisition will be applicable during its processing. A parse tree for a specific input program of this language only captures the syntactical aspects of the program. The semantic information can be computed from that tree with the help of acquisition. Here are a few examples, all taken from the world of C++:

**Constructors** Constructors and ordinary member functions have essentially the same syntax, but generate quite a different code. The precise type of method can only be
determined by the name property of the enclosing class/struct definition.

**Variable definition vs. function declaration** The \( x \ y(z); \) \( C^{++} \) statement is either a function declaration or a variable declaration, depending on the declarations of \( x, y \) and \( z \). These declarations can only be found in the environment surrounding the statement.

**Members’ visibility** The visibility (public, protected or private) of members in an aggregation is also determined by the type of the aggregation, struct or class.

**Contexts** More generally, each scoping unit: file, function, class, struct or namespace, acquires a context from its enclosing scoping unit, may override this context in part or in whole, and passes on the modified context to elements enclosed in it.

In summary, as is the case in life, things must be put in context. Observe that in the last two items above, an enclosed element not only acquires properties of the enclosing one, but may also change them: For example, a protected: statement in a struct changes the visibility attribute of its enclosing struct. We deal briefly with this delicate issue later.

## 3 Requirements from an Environmental Acquisition System

After the need for acquisition has been motivated, we can turn to the task of deriving requirements from a system (e.g., a programming language, a database engine, etc.) that supports it. Elementary, a system that supports acquisition must support a notion of environment, which we define as the list of enclosing composites of a component. Our first requirement ensures just that.

- **Requirement 1 (Recognizability of composition):** *Composite-component relationships among objects must be distinguishable from all other relationships.*

Composite-component relationships here are understood in their usual sense: An object may be a component of (directly contained in) at most one composite, and no object may be contained, directly or indirectly in itself. A composition root is an object which is not a component of any other object. Other relationships here are, for example, attribution as in Car-Colour, or various associations as in Car-Owner. Acquisition should occur only along links which represent composite-component relationships and no others.

As simple as the above requirement may seem, it is not satisfied by most programming languages, including SMALLTALK which adheres to reference semantics, including mixed
value and reference semantics languages such as C++, and including new versions of Eiffel [32]. In reference-semantics languages, there is no clear distinction between containment and other kinds of associations. With value semantics (e.g., Eiffel’s expanded), an object is contained in another if its value is stored in it. However, pure value semantics complicates the implementation of very large wholes and wholes with a variable number of parts. This is hardly surprising, since even in the design and analysis stages, “...the difference between Whole-Part Associations (WPAs) and other associations is often only cosmetic and diagrammatic.” [8].

To continue the quote “While it is generally acknowledged that WPAs bind classes more strongly than other associations, there are no further rules or constraints to guide design and implementation decisions”. Our first requirement trivializes this implementation decision dilemma. Further, acquisition may also alleviate the problem of recognizing WPA from other kinds of associations in the design phase. This is because WPA translates into distinguished links, along which properties may propagate.

Note, however, that the term *whole* in the acronym WPA does not coincide completely with our understanding of the composite notion. Civello [8] suggests classification of wholes as *assemblies* in which WPAs are *functional*, or as either *aggregates* or *tuples* in which both WPAs are *non-functional*. We believe that acquisition should occur only along functional WPAs. However, it should also occur along *spatial or temporal inclusions* which are not WPAs according to Civello’s taxonomy.

Let us leave, beyond the scope of this research, the development of a complete theory characterizing the kinds of associations along which acquisition should occur. We content ourselves with the formal requirement that there are distinct hierarchical links among objects along which acquisition might occur. This requirement is complemented with the intuition built upon the motivating examples of the previous section.

As much as composite-component links are important, acquisition makes very little sense if the components cannot be accessed from outside the composite. After all, if the composite is the only one which can access its components, then there is no great need for it to pass on its properties to them.

- **Requirement 2 (Components’ visibility):** Composite objects must be able to export the ability to access the components they enclose as autonomous objects. *

---

8However, it is claimed by BETA designers that values capture containment, while references represent other kinds of association.

9Perhaps surprisingly, we can make the marginal note that it is not essential that the composition root is accessible at all. One can imagine a system in which there is a single composition root, and that no direct access to it is required.
This requirement may be viewed as a violation of encapsulation. However, a “guilty” verdict in this charge will also implicate virtually all\(^\text{10}\) other OO systems in which an encapsulating component may still return, as a response to a method, a handle to one of its components. Quite often the “secrecy” of the structure of a composite should be relinquished as part of its interface. Cases in point are the application domains enumerated in the previous section: access to components and sub-components, and even the traversal of the entire composition tree, are the rule rather than the exception. What is really meant by encapsulation is that although a composite may reveal its components in order to allow direct access to them, such a revelation is not essential for accessing the object itself.

- **Requirement 3 (Encapsulation of components):** *The interface for accessing a composite must not depend on knowledge of its components.*

This requirement should actually be interpreted as a principle of good definition of an interface in any OOP system, together with the requirement that the system support such definitions. Of course, these are hardly new\(^\text{38}\) and are rather easy to implement. It is much more challenging to meet the dual requirement in an acquisition system:

- **Requirement 4 (Encapsulation of environment):** *The interface for accessing a component must not depend on knowledge of its environment.*

It might be argued that this requirement is not vital. A client that holds a handle of a component must have acquired it through descent in the composition chain. The surrounding environment of a component is therefore exposed to clients. However, exploitation of this knowledge is undesired since:

1. it is complicated devise the exact protocol for each one of the unbounded different possible environments;
2. it induces strong coupling among three agents: the environment, the component and the client; and
3. variable protocols conflict with the spirit of class-based languages.

Consider a hypothetical implementation of acquisition in which all objects have an instance variable `composite` storing a reference to its immediate enclosing composite, if it exists. Suppose also that the standard binding of messages to methods is altered so that a message, which is not recognized by a receiver, is passed on to its `composite`. This could be done in a *forwarding* manner, i.e., method execution in the context of the `composite`, or in a *delegation* manner, that is method execution in the context of the original receiver. (In the

\(^{10}\)All other OO systems with the exception of perhaps only Hogg’s Islands\(^\text{25}\).
following section it is explained why forwarding is preferred over delegation for acquisition, but this distinction is of secondary importance here.)

This simple mechanism and the simple method of describing it, pose an implementation challenge in statically-typed, compiled languages such as C++ and calls for sophisticated tricks [10, Section 9.2]. It is very feasible, however, in modern Smalltalks and in other dynamically-typed, interpreted languages. Nevertheless, the sort of acquisition it provides is not adequate since Requirement 4 is not met. The sender of a message must be familiar with the concrete environment as determined in receiver’s runtime state in order to know if the messages will be recognized or not. This is also the source of the difficulty in an implementation in C++-like languages: the mechanism prescribes that the protocol of an object can only be determined in run-time.

A corollary of Requirement 4 is therefore:

- **Requirement 5 (Environmental static properties):** All environmentally inherited properties that are part of the interface of a component should be specified in the class definition of this component.

This merely restricts the problem of dealing with the unbounded number of very different environments to the time of class definition without making progress towards its solution. What is needed is:

- **Requirement 6 (Static environmental properties):** At the time of class definition, it must be possible to infer, in an orderly manner, the potential environments which may surround objects of this class.

4 Static Analysis with Acquisition

In dynamically-typed languages, any instance variable of any class may contain any other object(s). It is therefore impossible to infer (or prove) statements such as “objects of class C will never be contained in objects of class C’”. In the general case, this is undecidable. Partial inference may be possible, but it is not clear to what extent the results it may generate are complete enough to be of use. The static-typing alternative is much more appealing in this respect and, as explained above, brings about some interesting challenges.

In a good statically-typed acquisition system we expect to see the abilities of declarations of:

1. The usual OOP concepts of classes, properties, inheritance, messages, methods, etc.

2. Typed components of objects of a class (henceforth called *slots*), which are distinct from the attributes of objects of the class.
3. Required slots, optional slots, and slots which may occupy a list of components of the same type.

4. Classes which must, may or never be components of other classes.

### 4.1 Basic definitions

**Definition 4.1 (component-composite)** Let \( a \sqsubseteq b \) denote the relation “\( a \) is a direct component of \( b \)” defined over instances. Let \( \sqsubseteq^* = (\sqsubseteq)^* \) denote its transitive closure.\(^{11}\)

For two arbitrary classes \( C \) and \( C' \), the system must be able to prove predicates such as

1. \( \forall a : C \cdot \exists b : C' \cdot a \sqsubseteq b \).
   “Objects of \( C \) can only occur as direct components of objects of class \( C' \)”

2. \( \forall a : C \cdot \exists b : C' \cdot a \sqsubseteq^* b \).
   “Objects of \( C \) can only occur as direct or indirect components of objects of class \( C' \)”.

3. \( \exists a : C \cdot \exists b : C' \cdot a \sqsubseteq b \).
   Together with the next predicate: “objects of \( C \) may or may not occur as direct components of objects of class \( C' \)”.

4. \( \forall a : C \cdot \forall b : C' \cdot a \not\sqsubseteq b \).

The quantifiers in the above should be interpreted as follows: \( \forall a : C \cdot P(a) \) means that in all plausible runs consistent with the type declarations, the predicate \( P(a) \) holds for all objects of class \( C \), whereas \( \exists a : C \cdot P(a) \) means that there is a run consistent with the type declarations in which \( P(a) \) holds for some object \( a \) of the class \( C \). Of course it should be defined precisely what are runs and which runs are “consistent with the type declarations”. In this extended abstract, we count on the intuition of the reader, experienced with typeful programming, to make up for this omission.

Observe that the above predicates are actually relations over *classes*. The first predicate can be expressed in terms of the others; The second will be given a special notation \( C \prec C' \) in Definition 4.3; The last two predicates are denoted in Definition 4.2 as \( C \sqsubseteq C' \) and \( C \not\sqsubseteq C' \) respectively.

**Definition 4.2 (constituent-construction)** A class \( T \) is said to be a direct constituent of \( T' \), denoted \( T \sqsubseteq T' \), and \( T' \) is said to be a direct construction of \( T \), if instances of \( T \) may occur as direct components of an instance of \( T' \).\(^{12}\) Let \( \sqsubseteq^* = (\sqsubseteq)^* \) denote the transitive closure of \( \sqsubseteq \).

\(^{11}\)No need to take inheritance into account (yet) because ‘\( \sqsubseteq \)’ is so far defined over objects, not classes.

\(^{12}\)I.e., some ancestor of \( T \) is a slot in some ancestor of \( T' \); \( \exists a : T \cdot \exists b : T' \cdot a \sqsubseteq b \).
Let $T \in T'$ denote the relation over classes “a slot of type $T$ is directly defined in $T'$”. The relation ‘$\in$’ is not necessarily reflexive and its closure (over classes) has no useful meaning because it does not take subtype polymorphism into account. Definition 4.2 of constituent-construction (‘$\subseteq$’ over classes) is actually the closure of ‘$\in$’ with respect to inheritance. Note that we ignore at this stage the issue of abstract classes and other less trivial cases where a given class never instantiates objects. Also note that there is a difference between “may occur”, “may-only occur” and “may-and-may-only occur”, and that the negation of “may-not” is not “may-only occur”. Definition 4.3 captures the useful notion of “may-only occur”:

**Definition 4.3 (resident-residence)** A class $T$ is said to be a resident of $T'$, denoted $T \prec T'$, and $T'$ is said to be the residence of $T$, if instances of $T$ must occur as direct or indirect components of $T'$.

### 4.2 Type inference

In a static type system, inferences such as constituent and resident are easy to implement algorithmically although sub-classes make this implementation more tricky. Requirement 1 states that Definition 4.2 (constituent-construction) is distinguishable from all other relations on classes. Definition 4.3 is required to keep Requirement 6. It can be shown that the inference system is sub-recursive, that is, the predicates we are interested in can all be inferred within the framework of first order predicate calculus augmented by transitive closures. Further, we can state:

**Theorem 1** Given a set of class declarations, there is an algorithm which in time polynomial in size of the input, will compute the truth value of the following four predicates (written in abridged form) for every two classes $C_1$ and $C_2$ in the system: “instances of $C_1$ must/may be a direct/indirect component of instances of $C_2$”.

In order to prove theorems such as the above (and comply with Requirement 6), we will describe a little more formally how acquisition interacts with inheritance, and we will provide (in Lemma 4.1) a precise algorithm for the predicates of Theorem 1.

Let $T \leq T'$ denote the relation “$T$ is a direct descendant of $T'$”, and let the transitive closures be $\leq^* = (\leq)^*$. Inheritance is a relation on classes. Practically it affects objects, of course: (descendant $\leq$ ancestor) implies (subobject of type ancestor $\subseteq$ (object of type descendant)). Composition, on the other hand, is the opposite. It is basically a relation on instances that (statically) imposes a relation on classes: (component $\subseteq$ composite) implies (class of component $\subseteq$ (class of composite)). A summary of the terms and notations defined is brought in Table 1.
Table 1: Terms and notations of the defined relations.

<table>
<thead>
<tr>
<th>relation</th>
<th>classes \times \text{classes}</th>
<th>over</th>
<th>instances \times \text{instances}</th>
</tr>
</thead>
<tbody>
<tr>
<td>declared in</td>
<td></td>
<td>\text{slot-in}</td>
<td>slot \in \text{board}</td>
</tr>
<tr>
<td>direct inheritance</td>
<td></td>
<td>\text{child-of}</td>
<td>descendant \leq \text{ancestor}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>descendant \preceq \text{ancestor}</td>
</tr>
<tr>
<td>direct composition</td>
<td></td>
<td>\text{may-occur}</td>
<td>constituent \preceq \text{construction}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>constituent \preceq \text{construction}</td>
</tr>
<tr>
<td>(must reside)*</td>
<td></td>
<td>\text{must-occur}</td>
<td>resident \prec \text{residence}</td>
</tr>
</tbody>
</table>

* denotes the transitive closure.

**Lemma 4.1** The relation “may-occur” (‘\(\sqsubseteq\)’) could be efficiently constructed from the relations “slot-in” (‘\(\in\)’) and “child-of” (‘\(\leq\)’).

**Proof:** The relations ‘\(\sqsubseteq\)’, ‘\(\in\)’ and ‘\(\leq\)’ are all subsets of the Cartesian product \(C \times C\), where \(C\) denotes the set of classes. Assume these relations represent an adjacency matrix of a directed graph. Given a directed graph with two sets of labeled edges \(G = (C, \{R_\sqsubseteq, R_\leq\})\), the algorithm outputs the graph \(G' = (C, R_\sqsubseteq)\) in time polynomial in \(|C|\). By definition, \(T\) is a direct constituent of \(T'\), \(T \sqsubseteq T'\), if and only if some ancestor of \(T\) is a slot in some ancestor of \(T'\), i.e., \(\exists S, S'\) such that \(T \preceq S\) and \(S \in S'\) and \(T' \preceq S'\). Hence, we can construct \(R_\sqsubseteq\) as follows:

1. For every \(T, S\) in \(C\),
   (a) If \((T \in S)\) is in \(R_\in\), add the edge \((T \sqsubseteq S)\) to \(R_\sqsubseteq\).
   “A slot of \(S\) is also a constituent of \(S\”).

2. While new edges are added to \(R_\sqsubseteq\) do for every \(T, T', S, S'\) in \(C\),
   (a) If \((T' \sqsubseteq S)\) is in \(R_\sqsubseteq\) and \((T \preceq T')\) in \(R_\leq\), add the edge \((T \sqsubseteq S)\) to \(R_\sqsubseteq\).
   “A descendant of a constituent of \(S\) is also a constituent of \(S\”).
   (b) If \((T \sqsubseteq S')\) is in \(R_\sqsubseteq\) and \((S \preceq S')\) in \(R_\leq\), add the edge \((T \sqsubseteq S)\) to \(R_\sqsubseteq\).
   “An constituent of an ancestor of \(S\) is also a constituent of \(S\”).

To show that this construction may be done in polynomial time we rely on transitive closure of graphs. The transitive closure of a given graph \(G = (V, E)\) is \(G^* = (V, E^*)\), where
between every pair of vertices $i,j$ in $V$ there is a directed edge $(i \to j)$ in $E^*$, if and only if $E$ has a directed nonempty path $(i \leadsto j)$. Assume $R$ is given as an $n \times n$ matrix, $(R_{i,j})$, over a closed semi-ring $\langle \{0,1\}, \vee, \wedge, 0, 1 \rangle$ [1], where `$\vee$' is the binary summary operator and `$\wedge$' is the binary multiplication operator, and $R_{i,j} = 1$ iff $(i \to j)$ iff $iRj$. Then $[11]$, $R^* = \sum_{k=1}^{\infty} \prod_{i=1}^{n} R_i = \bigwedge_{i=1}^{n} R_i$. Note that if $R$ is reflexive ($R_{i,i} = 1$ for all $1 \leq i \leq n$), then $R^* = R^n$. Given $\langle R_\varepsilon, R_\leq \rangle$, we can construct $R_\bullet$ as follows:

$$R_\bullet = R_\varepsilon \lor R_\leq \bullet R_\leq R_\bullet^T \vDash = R_\varepsilon^* R_\varepsilon \left( R_\leq^T \vDash \right) = R_\leq^* R_\varepsilon \left( R_\varepsilon^T \right)^T$$

and this can be computed in $\Theta(\|C\|^3)$ and with efficient matrix multiplication in $O(\|C\| \cdot \|C\| \cdot \lg \|C\|)$, where $O(\|C\| \cdot \|C\|)$ is the time complexity of multiplying two $\|C\| \times \|C\|$ matrices, $\gamma \approx 2.379$. □

So far we dealt with the "may" part of Theorem 1 and not with the "must" – "can only occur" part. At this point the reader may wonder what does it mean to possibly have a construction, and how is it possible at all to guarantee having one. We assume, what we feel is a common hidden assumption, that a class can be declared as a pure constituent, in which case it can only be instantiated as part of a composite or as a pure construction, in which case it can never be instantiated as a component of an object of another class.

These properties are inherited and cannot be overridden in inheritance. A class for which no such declaration is made can serve as either a constituent or as a construction. Classes inheriting from such a class may be declared as pure constituents or as pure constructions.

Let the pseudo class $\text{World}^{14}$ represent the "ultimate construction of all constructions", and let $T \in \text{World}$ denote that $T$ is not declared a pure constituent. Then, $T \sqsubseteq \text{World}$ means that an instance of $T$ may be a composition root (an instance that is no one's component).

**Corollary 1 (pure constituent, pure construction)** A class for which $\text{World}$ is its only direct construction is a pure construction. A class for which $\text{World}$ is not its direct construction is a pure constituent.

**Example** As a simple example consider the structure of a LaTeX document. Since the root of a document is always Document, any Section is guaranteed to have a construction of type Document. On the other hand, Section cannot exist but as a constituent of Document. Thus Document should be declared a pure construction, and Section should be declared a pure constituent.

```
World {
    Document;
    ...;
};
```

---

13 Read: it must be provable by static analysis only that...
14 This can coexist with "Any" that is sometimes used to denote the root of the inheritance trees.
Note that the distinction between pure constituent and construction classes with respect to acquisition is at the same level as abstract and concrete classes are with respect to inheritance. We can now use quantifiers over the domain of all classes, denoted here $\mathcal{D}_C$, to express predicates such as:

1. $C \not\subseteq \text{World}$
   “Instances of $C$ can only occur as components of other objects”.

2. $\forall C' : \mathcal{D}_C \cdot C \not\subseteq C'$
   “Instances of $C$ cannot occur as components of other objects”.

3. $C \subseteq \text{World} \land \forall C' : \mathcal{D}_C \cdot C \not\subseteq C'$
   “Instances of $C$ may and may only occur as roots”.

4. $\forall C' : \mathcal{D}_C \cup \{\text{World}\} \setminus \{C_2\} \cdot C_1 \not\subseteq C'$
   “Instances of $C_1$ may only occur as a direct component of an instance of $C_2$”.

5. $C_1 \prec C_2$
   “Instances of $C_1$ may only occur as a direct or indirect component of an instance of $C_2$”.

6. $\forall C : \mathcal{D}_C \cdot C \prec \text{World}$
   The tautology: “Every instance must be a direct or indirect component of World”.

7. $\forall C_1, C_2 : \mathcal{D}_C \cdot C_1 \prec C_2 \implies C_1 \subseteq C_2$
   “If a class must be a constituent of another class, then it may also be a constituent of that class”.

8. $\forall C_1, C_2, C_3 : \mathcal{D}_C \cdot C_1 \prec C_2 \land C_2 \prec C_3 \implies C_1 \prec C_3$
   Transitivity of ‘$\prec$’: “‘$\prec$’ is closed under transitivity”.

**Lemma 4.2** The relation “must-occur” (‘$\prec$’) could be efficiently constructed from the relation “may-occur” (‘$\subseteq$’) on the set of classes including the pseudo class World ($\mathcal{D}_C \cup \{\text{World}\}$).

*Proof:* Given a directed graph with two sets of labeled edges $G = (\mathcal{C}, \langle R_\subseteq, R_\prec \rangle)$, the algorithm outputs the graph $G' = (\mathcal{C}, R_\prec)$ in time polynomial in $|\mathcal{C}|$. It is sufficient to show that $R_\prec$ can be obtained by an efficient matrix multiplication. The relation “must-occur” is a
subset of the relation “may-occur”: A class must be allowed to occur in order for it to have to occur. This subset is recursively obtained by the following construction:

1. Direct occurrences: A must (directly) occur in B if A may directly occur in B and all the classes, that A may directly occur in, are direct or indirect descendants of B. This includes B itself since a class is considered to be a descendant of itself. That is, \( A \prec B \) if

\[
A \subseteq B \quad \text{and} \quad \forall C : C \bullet (A \subseteq C) \implies (C \leq^* B)
\]

and generally for any two classes \( C_i, C_j \in \mathcal{C} \), we can state that \( C_i \prec C_j \) if

\[
C_i \subseteq C_j \quad \text{and} \quad \forall k, 1 \leq k \leq |\mathcal{C}| : (C_i \subseteq C_k) \implies (C_k \leq^* C_j)
\]

\[
C_i \subseteq C_j \quad \text{and} \quad \bigwedge_{k=1}^{\lvert \mathcal{C} \rvert} [[C_i \not\subseteq C_k] \lor (C_k \leq^* C_j)]
\]

\[
C_i \subseteq C_j \quad \text{and} \quad \lnot \bigvee_{k=1}^{\lvert \mathcal{C} \rvert} [(C_i \subseteq C_k) \land (C_k \not\leq^* C_j)]
\]

Let \( \overline{R} \) denote the negation of \( R \). Using matrix multiplication this is expressed as

\[
R^{(1)}_\prec = R_E \wedge \overline{R_E R_{\leq^*}}
\]

2. Induction step: A must (indirectly) occur in B if A may directly occur in B and all the classes, that A may directly occur in, must (indirectly) occur in B.

\[
R^{(n)}_\prec = R_E \wedge \overline{R_E R_{\leq^*}^{(n-1)}}
\]

By induction we get

\[
R_\prec = R^{(|\mathcal{C}|)}_\prec = \left( R_E \wedge \overline{R_E \left( \cdots \left( R_E \wedge \overline{R_E R_{\leq^*}} \right) \cdots \right)} \right)
\]
4.3 Reasoning about acquisition

For the sake of simplicity, assume that a class may define only simple features which are constant values associated with an object from birth to death. Nonetheless, a feature’s value may be subjected to environmental effects, as shall be explained. Also for simplicity, assume both strict inheritance and strict acquisition. In order for a feature to acquire a value, construction need rely on the context the object resides in, i.e., its environment. Object construction becomes context-sensitive in some sense. Some objects may be limited to a specific context. In other cases objects of the same class may exist in different contexts, (possible residing with different values to their environmental features). Some contexts may be outside the limit for some classes while suitable for others. Thus environmentally-affected constant-features are indeed possible.

Clearly a feature’s value is part of the particular properties that differ in different objects of the same class. The challenge (and novelty) is in having their existence as well as their type known a priori as part of the shared properties which are determined by the object’s class (recall Requirement 5). Questions that need to be addressed: Is such a model reasonable? Why is it important to know existence and type at class level? These questions are addressed later, but first let us make it clear what we expect from such features.

A class may declare some of its attributes as acquisition from its constructions in the containment tree. The type containment inference algorithm verifies that all such usages of type acquisition are legal. That is, if a class \( C \) acquires an attribute \( x \), then it must be provable that there is a class \( C' \) such that \( C' \) has an attribute \( x \) and that instances of \( C \) are always contained, directly or indirectly, in instances of \( C' \). It is possible also to use an optional attribute \( x \). In which case, it must only be provable that the instances of \( C \) may occur in instances of \( C' \). In this case, the definition of \( C \) must provide some default value for the case that no object of the class \( C' \) is found among the constructions. There are similar mechanisms that apply for referring to the (potentially empty) list of attributes \( x \) in all of the containers in which it is defined.

In the definitions of its attributes a class may also refer to other dynamically inherited/acquired attributes, but without exporting them. (Currently, we ignore support for fancier encapsulation.) It is also possible to refer in this way to a composite object. All such referrals are naturally subject to verification by the type containment inference algorithm.

The compiler can possibly examine all acquisition references from a class and then define hidden pseudo-slots which contain pointers of an appropriate type to all constructions. It may also produce an appropriate code for filling these pointers at instantiation time. This code can be efficient; all slots can be filled in a simple two-pass scan of the instantiation tree.

Acquisition of optional and mandatory properties is closely associated with the may and

\footnote{read: both “may” and “may not” occur}
the *must occur in* relations, respectively. However, there is no complete correlation between the two. A class may acquire properties from classes it may (must) occur in as well as from classes in may (must) never occur in. In Figure 5, for example, *Auto_part* can *never* occur in the abstract class *Vehicle*, but the environment of *Auto_part* *always* has a *driver* inherited from *Vehicle*. That is, *Auto_part* and *Vehicle* are two classes that are not directly related, but a property defined in the latter is environmentally safe for the first. The same can be true with optional properties. If *Auto_part* is put in a *Sport_car*, it may acquire the *skill* of car racing despite not ever being a component of *Sport*.

![Figure 5: Class *Auto_part* acquires properties from classes it may occur in as well as from classes it may never occur in. *Auto_part* may acquire properties from *Car* and *Sport_car* (e.g., *colour* from *Car* and *sponsor* from *Sport_car*) as well as acquire properties from their ancestors which may be (abstract or other) classes in which *Auto_part* may never occur (e.g., *driver* from the abstract class *Vehicle*, *skill* from the abstract class *Sport*).

Moreover, not all properties acquired, even through a “*must occur*” relationship, are necessarily safe. For example, in Figure 6, *Auto_part* must occur in *Car* (since it is itself an abstract class and it may not occur in any other class but *Car*). But knowing that “*Auto_part* must occur in *Car*” is not enough for deducing that *Car*’s properties, like *driver*, are safe for *Auto_part* to acquire. *Auto_part* indeed acquires the property *driver*. However, a *variables* of type *Auto_part* might hold an object of type *Door*, and *Door* does not have to be part of a *Car*.

The interpretation of \( C \prec \hat{C} \) is that properties of \( \hat{C} \) are environmentally safe for *objects* of (dynamic) type \( C \). Our new terminology and notation allows us, nevertheless, to formulate extra requirements. We can require, for instance, that static safety should imply dynamic safety, by stating:

\[
\forall C, C', \hat{C} : D_c \bullet (C \preceq^* C' \text{ and } C' \prec \hat{C}) \Rightarrow C \prec \hat{C} \quad (1)
\]
Figure 6: Class Auto_part must occur in Car, but its descendant Door may occur in either Car or Airplane. Thus, property driver is not safe for variables of (static) type Auto_part, because if its run-time (dynamic) type is Door it may occur in an Airplane that has a pilot instead.

That is, if a system satisfies condition (1) then any property that is safe for a static type is also safe for all its possible dynamic types. Alternatively, if a system does not satisfy (1), we can define a sub-relation of ‘←’ that satisfies the condition, and apply that new relation to deduce the type-safe properties.

Another example is condition (2) that overrules coincidence safety:

$$\forall C : D_c \bullet \exists \hat{C}' : D_{\hat{C}} \bullet \forall \hat{C} : D_{\hat{C}} \bullet (C \sqsubseteq \hat{C}) \iff (\hat{C} \leq \hat{C}')$$  \hspace{1cm} (2)

Condition (2) forces a representative of the set of potential environments to be an ancestor of the rest. Thus making it both a residence and a platform for placing all safe properties. Hence, any case of coincidental environmental overloading can be changed to safe environmental overriding. The next section elaborates this notion by comparing the concepts of inheritance with those of acquisition.

5 Concepts of Environmental Acquisition

In this section we try to gain some deeper understanding of acquisition by revising the fundamental terms of inheritance and finding analogous concepts and phenomena in acquisition.

Overriding The term overriding refers to the redefinition of an inherited property. In inheritance it is the descendant class that dominates properties inherited from its ancestors. In acquisition the default behaviour should be just the opposite of this. That is to say, the inherited property overrides the local property. Suppose that a class \( C \sqsubseteq C' \), but \( C \not\subset C' \), and that a feature \( f \) is defined in \( C' \). Then, it would not be type safe to invoke, by means of acquisition, the feature \( f \) from the class \( C \) unless \( f \) is defined also in \( C \). Assume that is the case. If \( C.f \) overrides \( C'.f \), then this means that \( C'.f \) can never be called safely, whereby making acquisition pointless in the very
common case of the ‘⊆’ relationship. Conversely, by letting \( C'.f \) override \( C.f \), we get interesting and useful semantics. For example, a black Door put in a white Car should turn white. But once removed it should show black again.

We now observe that there are cases in which a component needs the ability to override the composite. In our examples it should be possible to paint just the door green after it had been placed in a white car.

We propose the following operational model for overriding: each attribute in each object is associated with an override flag. This flag controls whether the component overrides the composite or vice versa for a specific property. When an attribute of a certain object is set to some value, the override flag turns true. In addition, the override flag of all components of this object turns false. By painting the car blue then, all doors become blue as well. By further painting a particular door of the same car green, the handle becomes green as the door, but the car as a whole remains blue. When a free object is moved to be part of a component, the override flag of all of its attributes turns false, thus supporting the black-white door colour flipping behaviour described above.

An alternative model could apply a priority value instead of a boolean flag, and support a polymorphic `undef` value for valueless features, that is overridden regardless of the override flag.

**Refinement** In inheritance a method can refine and extend an inherited property. We call this \( \alpha \)-type refinement to distinguish it from \( \beta \)-type refinement named after “in-refinement” using the `inner` keyword of BETA and SIMULA. With acquisition \( \beta \)-typed refinement it has to be statically proven that an inherited definition would always exist in the object executing the method. To streamline the definition of refined methods it is useful to introduce a new keyword, e.g., `composite`, which is the acquisition analog of SMALLTALK’s `super`. Method refinement is as abundant in acquisition as it is in inheritance. For example, the following method implements \( \LaTeX \)'s emphasized behaviour:

```plaintext
font emphasized()
{
  if (composite.emphasized == #roman)
    return #italic;
  return #roman;
}
```

Note that the binding of `composite` must be static, just as is the case for `super`.

**Overloading** The term `overloading` refers to the use of a single name for several different properties in a single scope. Let \( \text{defined}(f,T) \) denote the relation “a prop-
property \( f \) is directly defined in \( T' \). In such a case, \( T::f \) denotes the definition of \( f \).

Let \( \text{applicable}(f, T) \) denote the relation “a property \( f \) is applicable for instances of \( T' \).” A property must be defined in one of the ancestors of \( T \) in order to be applicable. In such a case, if \( a \) is an instance of \( T \) and \( \text{applicable}(f, T) \), then \( a.f \) denotes the value of applying \( f \) in the context of \( a \).

Note that \( \text{applicable}(f, T) \) iff \( \exists T'' \) such that \( \text{defined}(f, T'') \) and \( T \leq^* T'' \). However, in such a case note that \( T::f \) is not necessarily \( T''::f \). This is because it is possible that the definition of \( f \) in \( T'' \) is overridden in some proper descendant of \( T'' \), which is also a proper ancestor of \( T \), e.g., a class \( S \) such that \( S \not\equiv T' \) and \( T \leq^* S \leq^* T' \). It is not enough for a property to be applicable in all the possible constructions in order to be environmentally applicable. A property must be applicable in a residence in order to be environmentally applicable. Consider, for example, that a class \( A \) must occur in either \( B_1 \) or \( B_2 \). Assume now that a property \( f \) is defined for both \( B_1 \) and \( B_2 \). Presumably \( a.f \) should be allowed, since it is statically known that \( f \) would be dynamically applicable for any \( a \) of \( A \), because any instance \( a \) of \( A \) is guaranteed to be either part of an instance of \( B_1 \) or part of an instance of \( B_2 \).

\[
\forall a: A \bullet (\exists b_1: B_1 \bullet a \sqsubseteq b_1) \lor (\exists b_2: B_2 \bullet a \sqsubseteq b_2)
\]

Nevertheless we do not allow this. Even if \( f \) is applicable in all possible environments of \( A \) it does not necessarily imply that \( f \) is environmentally applicable for \( A \).

\[
(\forall C': D_C \bullet A \sqsubseteq C' \implies \text{applicable}(f, C')) \iff \text{envapplicable}(f, A)
\]

If, however, \( B_1 \) and \( B_2 \) are descendants of some class \( D \) residence of \( A \) and \( f \) is defined for \( D \), then indeed \( f \) is environmentally applicable for \( A \). This is true even if the definition of \( f \) in \( D \) is differed. Thus, we can state that:

**Corollary 2 (envapplicable)** A property \( f \) is environmentally applicable in \( T \), denoted \( \text{envapplicable}(f, T) \), if it is either applicable in \( T \) or applicable in its residence.

The reason for insisting that the property is applicable in a residence is that we wish to support environmental-overrideing but not environmental-overloading. This is similar to the case in multiple inheritance (in C++) where a property is not applicable if the super-classes overload its meaning, but it may override a property (even more than once) on any distinct inheritance path.

One conclusion derived out of the above is that this is a case in favor of Multiple Inheritance (MI). An interesting related theme is the distinction between interface inheritance and class inheritance and interface-class inheritance.
**Interface Acquisition** A class declares a protocol and a structure, it can extend some other class via single inheritance, and can implement an interface. An Interface declares only a protocol (without a structure), and can extend other protocols via MI. Direct acquisition possibly involves a path of inheritance links. With inheritance there are 3 common links [33]: subtyping, abstraction, and inheritance. With acquisition there are 4 kinds of links: interface acquires from an interface, class acquires from an interface, class acquires from a class, and class acquires from an interface-inherited-by-a-class.

We can further distinguish between may-acquisition and must-acquisition and take optionality and multiplicity into account.

**Multiple Acquisition** MI of inheritance is *multiple* environmental acquisition of acquisition. In *multiple* environmental acquisition an object is simultaneously a component of more than one construction. This is in great contrast to our intuitive understanding of composition. However, one may imagine having two independent composition hierarchies. For example, in a large professional organization, each individual may have both a professional and a task-wise association. ‘Or’-acquisition, the acquisition equivalent of ‘or’-inheritance, is the (common) situation where a class may be a constituent of more than one construction.

**Dynamic Acquisition** Dynamic inheritance is the ability of an object to dynamically change its class. In acquisition this is the ability to move an object from one construction to another. For example, a car Door can be moved from a Sedan to a Hatchback.

**Abstract, Concrete, and Pure Classes** In inheritance abstract classes cannot be instantiated but can only be used for inheritance. All objects are instantiated from *concrete classes*. The analogous concepts for acquisition are:

1. Classes that can only instantiate components, but never roots. These have the capacity to environmentally acquire from other classes.

2. Classes that can only instantiate roots, but never components. These can never environmentally acquire from other classes.

It seems natural to associate the first with “abstract” and the second with “concrete”. Since it is a good practice in inheritance not to inherit from concrete classes, we expect it to be good practice in acquisition to restrict all classes to category 1 or 2 as in the above, but we recognize that this nice property cannot always be achieved.

In a language grammar this is exactly the case. We can look at an OO class hierarchy as representing a grammar [17, 19], e.g., [18, INTERPRETER], and we can look at a grammar as representing an OO class hierarchy [21]. Either way, a class hierarchy’s root class, our pseudo class World, corresponds to a grammar’s starting symbol.

**Frozen Classes** Frozen classes are classes that cannot be further used for inheritance. The equivalent in acquisition are atomic classes, i.e., classes that have no components.
Singleton Classes In inheritance a singleton is a class that has a single descendant. In acquisition a singleton would be a construction with a single constituent.

Polymorphism In inheritance there are two kinds of polymorphism cases, namely internal “this”-polymorphism, pertaining to polymorphism in the class implementation and external “pointer”-polymorphism, pertaining to polymorphism of the class’s clients. In acquisition the term environmental polymorphism corresponds to the internal polymorphism.

Dynamic vs. Static Binding So far, we tacitly ignored the issue of the context of execution of environmentally inherited methods. There are two possible semantics corresponding to ordinary (static binding) and virtual (dynamic binding) function members in C++. This distinction is also known as, especially in the context of links between distinct objects, the quarrel between forwarding and delegation semantics.

Forwarding means that the context of the initiating object is forgotten during the execution of an environmentally inherited method. If it invokes another method, then the method invoked is that defined in object of the current execution thread. Delegation means that a search must be made for the method to be invoked starting in the original object and then moving up the composition tree. That is to say, an invocation of a method is always done as if it was called directly by the original object regardless of the place in which the currently executing method was defined.

Suppose that a method \( m_1 \) of \( C \) calls another method \( m_2 \). Then, in the forwarding semantics, all that is required is to guarantee that \( m_2 \) will be found in run time, is to check that \( m_2 \) is defined in \( C \). After a moments reflection one can see that this static check covers also delegation. The search path for \( m_2 \) also includes the object that executed \( m_1 \), and since this object belongs in \( C \) and it therefore has a method \( m_2 \).

We believe that acquisition encompasses a sophisticated polymorphism by itself and that there is no need in most cases to complicate it further via delegation. Consider, for example, the case of “inherited” attributes in attribute grammars. The semantics of attribute grammars is such that the computation of attributes is done in a local context and only then propagated. Note that attributes in attribute grammars represent values, not executable methods, and there is no significance to their context of execution, but it seems as if most acquisition inherited features are values, not code fragments. If delegation semantics is required, then in many cases it can be implemented by means of inheritance. For example, a method for computing the paragraph indentation level in a text processing system could read

```c
float parindent()
{
```
return left_margin() + 1inch;
}

Clearly, left_margin() must be computed within the context of the current text. This would be the case if we use inheritance for this method.

Notwithstanding, there could be cases in which delegation semantics may be required also for acquisition. For example, one may want to apply different algorithms for computing the paragraph indentation in footnotes and in ordinary text. For that reason, we do allow delegation semantics for acquisition as well and suggest that they both may play an important role. This is in contrast to the case of inheritance, in which the existence of non-virtual function members have no useful semantics, and their existence is just another symptom of the C++ design policy of not penalizing a user for a language feature he doesn't use.

6 Comparison to other models and related works

In this section we briefly compare acquisition to other related work. Our main criteria for comparison are the requirements as developed in Section 3.

We first explain why some common OO constructs do not constitute by themselves an acquisition system and then turn into comparing our work with recent advances.

6.1 State as it is recorded in instance variables

It is possible to implement acquisition using object state. An object may contain a pointer to its composite (immediate construction). By following the chain of pointers it is possible to implement all of the intricacies of the acquisition. Indeed, systems such as \TeX, InterViews, TurboVision and many attribute grammar compilers do exactly this kind of emulation. One may therefore argue that the state of an object, including the value of this pointer includes all information required for acquisition. We disagree with this claim because of the following two reasons:

First, as should be clear by now, acquisition is far from being trivial. Its inherent complexity cannot be covered up. Re-implementation, as done for example in the software systems mentioned, is bound to be complicated, inefficient, and in many cases lacks well defined semantics. It is instructive to consider the case of attribute grammars. Although there is a huge body of work on their efficient implementation, they have failed to become popular. We feel that this is partly due to the fact that the algorithms were not packaged in a well defined language construct, and the need to enumerate all “inherited” attributes puts a heavy burden on the user.
Second, we feel it is wrong to blur the boundary between objects by stretching the meaning of the term "state" to include external objects to which pointers are stored. A network of objects, including perhaps all objects of a system, could be thus made into a single object. Further, since in general pointers are not bidirectional, and since there might be variables pointing to various locations in the network, a complicated semantics would have to be called in to describe the complex relationship between a subobject which corresponds to connected component of a directed graph. Instead, we feel that if there is a consistent pattern of acquisition of properties from one object by another, then it is better to define an abstraction mechanism that captures this pattern. Acquisition, and in particular the overriding mechanism described above, gives a good balance between state change, the change undertaken by the object itself, and the environmental influence.

6.2 Inheritance

Inheritance is a relation between classes, whereas acquisition pertains to individual objects. Other reasons why inheritance cannot serve for acquisition are given in the following discussion of dynamic inheritance.

6.3 Dynamic inheritance

Here we take this term in its restricted sense—a sense also called "configurable inheritance" [22]: if a class $C$ is declared as inheriting from $C'$, then the sub-object of objects of $C$, which correspond to the base-class $C'$, may also be of a class $C''$ which also inherits from $C'$. Loosely, $C$ may also inherit at run time from any class $C''$ which inherits from $C$. Stroustrup [43, Section 12.7] reports on a C++ extension proposal accompanied by an implementation experiment to that effect. Had this proposal of "delegation", as it is called there, been accepted, it could have only approximate acquisition, falling short in several important ways:

1. Dynamic inheritance just as a static one, induces a subtype relationship. This is usually undesirable in acquisition: a component is not necessarily a subtype of its enclosing composite. Observe that private inheritance merely restricts the visibility of sub-typing (as suggested in [41]) but does not exclude it altogether.

2. There is usually an orthogonal hierarchy classifying the kinds of objects that may occur in the containment hierarchy. Mixing the two hierarchies at the cost of the complexities of multiple inheritance and at the risk of blurring the distinction between the two hierarchies is worse.

3. It is inherent to acquisition that an "inherited sub-object" (a composite) is shared by many objects (its inheriting components). Such sharing contradicts our usual understand-
standing of inheritance.

This latter hindrance is the main reason for the failure of this experiment.

6.4 Delegation

To emulate an acquisition in a delegation system one would, as another approximation, replace substitute containment links with delegating ones.

Several problems arise. The first, and perhaps the easiest, is that of delegation vs. forwarding. As mentioned above, both semantics are feasible and useful for acquisition. Clearly, adding support for forwarding to runtime systems, such as that of SELF, which already support delegation should be a simple task.

Another problem is that, by Requirement 1 above the containment links should be distinguishable from other links. This again could be done via relatively easy syntactical changes. Another change required is reversing the links directions so that the composite has slots for the components and not vice-versa. Although no delegation system that we know of implements this, such an addition does not seem too problematic.

The third, and most difficult problem: delegation systems, almost by nature, subvert strong typing. Beyond the usual benefits of strong typing, such as improved efficiency and reliability, we argued in Requirement 5 above that acquisition should be done in a strongly-typed manner. In this respect, our research can be viewed as an attempt to investigate strong typing of a restricted form of a delegation system. The more general problem should be the subject of another investigation.

6.5 Other related work

Blake and Cook [2] deal with the issue of support to containment hierarchy in OO languages. They compare part hierarchies with inheritance and delegation and provide a mini-taxonomy for kinds of containment relationships. They also identify the dilemma of the visibility of parts, and argue strongly that exposing the parts does not violate encapsulation. Their solution is different from our Requirement 2. Specifically they propose, implement and report on the lessons learned from a SMALLTALK extension in which the composite forwards messages to its components. Another aspect in which their work is different from ours is that they tacitly ignore the question of strong typing and optional components. In our model strong typing and optional “parents” play an important role. We were unable to determine if Blake and Cook’s work makes the distinction between part and attribute links.

Kim et. al [26] examine the question of composite objects in OO database systems. They also argue that the composition links should be distinguishable from other links. They
provide a formal definition of the data model of composite objects and show how they might be used in a database system by addressing questions such as locking and efficiency. One of their interesting contributions is the support for versions of composite objects. (This is similar to the work in Vesta [5].) As it is common in databases, it is assumed in their work that parts have external visibility. No treatise is given to the question of propagating properties inside the composite object. We consider using their results in testing our proposal on large inputs, such as legacy systems, which need to be reverse engineered.

Kristensen [28] argues for and proposes lingual mechanisms to support general complex associations, including special notations for containment. In this sense and in the strong ties his work has with formal methodologies of analysis and design, it is more general than ours. However, acquisition is much more powerful than a meager existence of a containment link.

7 Conclusions, Open Questions and Further Research

In this work we demonstrated the need for down propagation of attributes in containment hierarchies. We called this propagation “acquisition”. A conjugate term is “environmental polymorphism” which relates to the polymorphic nature of objects due to their respective environments. We developed a set of requirements for a system which should support these features; among other things we argue that the exposition of the structure of part hierarchies should not be regarded as a violation of encapsulation. We showed that a proper implementation should observe static typing rules. A framework for inferring possible direct and indirect containments was developed.

This research presented many questions (perhaps even more than it answered). Some topics for further research are:

1. Formal semantics for the newly proposed computation model.

2. Semantics and support for multiple environmental acquisition, an “environmental multiple inheritance”.

3. Extending this work to deal with inheritance of attributes across arbitrary association links.

4. Optimizing the type containment detection algorithm.

5. Optimizing the actual parents detection algorithm and providing an efficient implementation for dynamic changes in the structure of objects.

\[\text{16}^\text{In this respect, it should be compared to the work of Civello [8]}\]
6. Lessons from larger scale experiments.

7. Versions support for part hierarchies.

8. Mechanisms for recursive types and quantified polymorphism and for investigating the inter-relationships with environmental polymorphism/acquisition (recall that rich enough static type systems include dynamic typing as well).

Acknowledgment The research of the first author is supported by the Technion V.P. fund for promotion of research at the Technion.

References


[37] OWL manual.


