# Environmental Acquisition Revisited

Richard Cobbe Matthias Felleisen

College of Computer and Information Science
Northeastern University
360 Huntington Ave
Boston, MA 02115
cobbe@ccs.neu.edu

## ABSTRACT

In 1996, Gil and Lorenz proposed programming language constructs for specifying environmental acquisition in addition to inheritance acquisition for objects. They noticed that in many programs, objects are arranged in containment hierarchies and need to obtain information from their container objects. Therefore, if languages allowed programmers to specify such relationships directly, type systems and run-time environments could enforce the invariants that make these programming patterns work.

In this paper, we present a formal version of environmental acquisition for class-based languages. Specifically, we introduce an extension of the ClassicJava model with constructs for environmental acquisition of fields and methods, a type system for the model, a reduction semantics, and a type soundness proof. We also discuss how to scale the model to a full-scale Java-like programming language.

## Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*classes and objects*; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*object-oriented programming*

## General Terms

Languages, Design

## Keywords

environmental acquisition, object-oriented languages, object containment, object composition

## 1. INTRODUCTION

Programs often manipulate hierarchical containers or tree-shaped forms of data. For example, a PLT Scheme [5] program that uses the MrEd library [6] for implementing a graphical user interface nests panels inside of a window to arrange the basic GUI widgets. Or, a Java program that uses the XML library creates a tree representation from some text that it reads.

Indeed, the idea of using hierarchical data representations of this kind is so standard that the collection of object-oriented (program) design patterns [10] contains special patterns, like the composite and interpreter patterns, for designing an appropriate set of classes. As many people have observed [12, 15], however, patterns almost naturally suggest new constructs for programming languages. Specifically, if a language codifies a pattern with a new construct, the compiler and the run-time system can enforce the pattern's programming invariants or, alternatively, can signal violations as soon as they recognize them.

Consider the case of hierarchical data. Typically, the design of hierarchical data implies that each item has a unique container, if it has one at all. Using the equivalent terminology of nested containers, an item is inside of (at most) one container, which in turn may be inside of another container and so on. Based on this property, programs tend to compute properties of an item that are a function of the item's uniquely determined environment. That is, items *acquire* properties from their environments and compute with them. Of course, this only works if the program establishes, maintains, and uses the hierarchy properly at all times, including so-called "friend pointers" from an item to its container (also known as "two-way links" in the UML community [8, p. 155]).

Gil and Lorenz [11] recognized this problem of *environmental acquisition* and proposed constructs for specifying such relationships directly within programs.[1] Their proposal was informal; it neither specified a formal type system for environmental acquisition nor stated a soundness theorem for a specific model. Since then, environmental acquisition has found its way into Python [22, 9] as a part of the Zope project [16]. Also, the second author has used environmental acquisition for a systems administration project where computers, switches, and ethernet cards acquire policies and other network information from their containers [17].

In this paper, we resume and continue this language design experiment. Specifically, we present a typed version of envi-

[1] As Gil and Lorenz point out, an object's *environment* may incorporate many different kinds of related objects. We restrict ourselves, as they do, to the environment provided by an object's containers.


*POPL '05*, January 12–14, 2005, Long Beach, California, USA.

```
class JComponent extends Container // ...
{
  // ...
  public JRootPane getRootPane() {
    return SwingUtilities.getRootPane(this);
  }
  // ...
}

class SwingUtilities // ...
{
  // ...
  public static JRootPane getRootPane(Component c) {
    if (c instanceof RootPaneContainer) {
      return ((RootPaneContainer)c).getRootPane();
    }
    for( ; c != null; c = c.getParent()) {
      if (c instanceof JRootPane) {
        return (JRootPane)c;
      }
    }
    return null;
  }
  // ...
}
```

Code from Java's Swing API [20].

**Figure 1: root panes in Swing**

ronmental acquisition for a class-based object-oriented programming language. Our goal is to show language designers how environmental acquisition helps programmers; what it takes to type constructs for environmental acquisition; and when the run-time system needs to check the invariants for acquisition relationships. A formal model, based on ClassicJava [7], provides the framework in which we discuss the design decisions and in which we can prove a type soundness theorem. Finally, we use the model to formulate a challenge for the designers of type systems concerning the elimination of run-time checks.

The paper consists of seven sections. The next section introduces environmental acquisition with concrete examples. Then the third section briefly recalls those elements of ClassicJava that matter for this paper. The fourth section is dedicated to Jacques, an extension of ClassicJava with constructs for specifying the environmental acquisition of fields and methods, its type system, and its formal semantics. In the fifth section we discuss several decisions concerning the design of Jacques. The sixth section sketches how to scale our model to a full object-oriented language. The seventh section sketches related efforts, their limitations, and how we might incorporate their methods into future versions of Jacques. The final section suggests some future work, especially on an improved type system for keeping track of acquisitions.

## 2. MOTIVATING EXAMPLES

As environmental acquisition relies on an object containment hierarchy, it has a strong connection with design patterns where such hierarchies play central roles, particularly the composite pattern and its close relative, the interpreter pattern. The focus of these patterns is often propagating information towards the root of an object tree. Acquisition

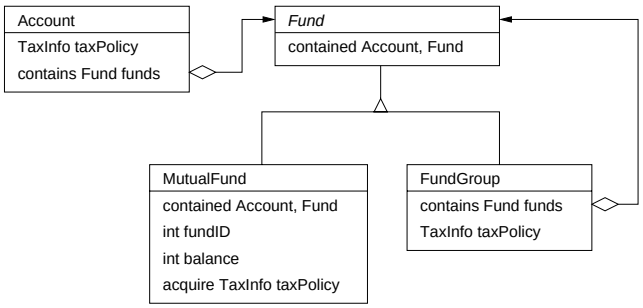



**Figure 2: acquisition example: investment accounts**

becomes relevant if the application must propagate information *away* from the root.

For example, consider GUI development toolkits, which are typically implemented with the composite pattern. In Java's Swing toolkit [21], for instance, all graphical components are subclasses of JComponent; several components like JPanel may contain other components. Every operating-system-level window in a Swing program has a single root pane; this pane contains the window's menu bar and other capabilities. Therefore, components that need to access the menu bar must have access to their containing root pane. Currently, the JComponent.getRootPane method retrieves the root pane by chasing explicit pointers up the panel containment hierarchy, as shown in figure 1. As GUI components are added, destroyed, and moved, the library maintains these pointers. With environmental acquisition, each individual component could simply acquire a reference to the root pane from its container, avoiding the need to write error-prone pointer-management code directly and instead leaving that responsibility to the run-time system.

Following Gamma et al, we take our next example of acquisition in the composite pattern from the financial sector. See figure 2 for the class diagram. The Account class represents an individual's account with a mutual fund firm; each account contains several *Fund* instances. It is frequently the case that an investor may have several types of funds within the same account; some funds may be standard investments, while others may be part of an IRA. To keep these funds separate for tax purposes, it is convenient to group multiple funds into a FundGroup instance. Now, transactions on a particular fund must take the fund's tax policy into account. With environmental acquisition, we simply declare the policy for the account as a whole, allow fund groups to override that policy for their funds, and have each fund acquire the policy from its nearest container.

For a final example, extracted from DrScheme [4], consider a class-writing wizard for use in a program development environment; see figure 3 for a UML diagram of the relevant classes. This wizard allows the user to define a new class or set of classes by specifying their properties graphically; the produce method in the ClassUnionWizard class generates the corresponding declarations in the target programming language. The system allows the user to define classes in several different manners. For example, the ClassInfo wizard constructs a single class, and the UnionInfo wizard constructs a sum-of-products type, as seen in the interpreter pattern.

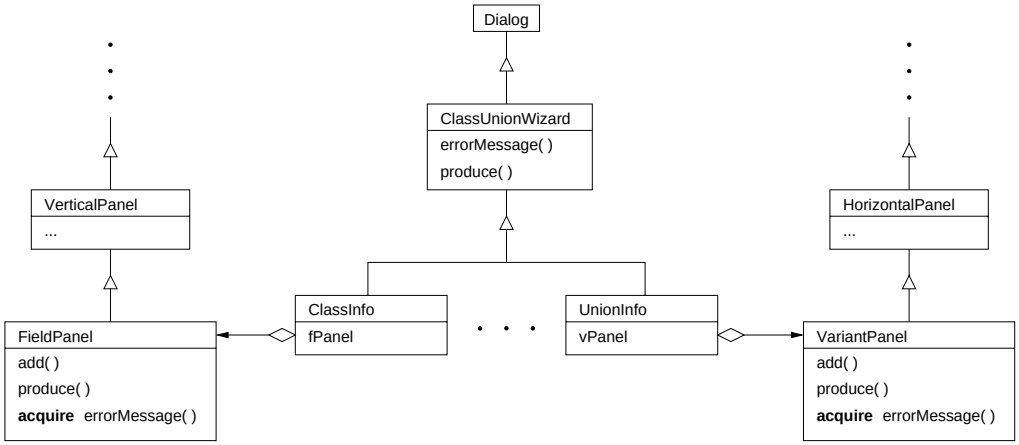
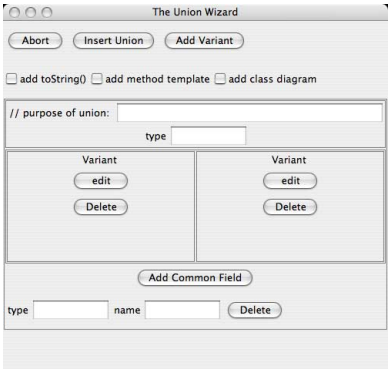



**Figure 3: acquisition example: class wizard**




Control interface

Work area

Error window

**Figure 4: class wizard dialog**

$$
\begin{array}{lcl}
P & = & defn^*\ e \\
defn & = & \mathsf{class}\ c\ \mathsf{extends}\ c\ \{field^*\ meth^*\} \\
field & = & t\ fd \\
meth & = & t\ md(arg^*)\ \{e\} \\
arg & = & t\ var \\
e & = & \mathsf{new}\ c \mid var \mid \mathsf{null} \mid \underline{e : c}.fd \mid \underline{e : c}.fd = e \\
 & \mid & e.md(e^*) \mid \mathsf{super}\ \underline{\equiv \mathsf{this} : c}.md(e^*) \\
 & \mid & \mathsf{cast}\ t\ e \mid \mathsf{let}\ var = e\ \mathsf{in}\ e \\
var & = & \text{a variable name or } \mathsf{this} \\
c & = & \text{a class name or } \mathsf{Object} \\
fd & = & \text{a field name} \\
md & = & \text{a method name} \\
t & = & c \quad \text{(a type name)}
\end{array}
$$

**Figure 5:** ClassicJava **syntax**

Each dialog contains three main visual components: a control interface at the top, an error window at the bottom, and a large work area in the center that is specialized to the specific wizard; see figure 4. For example, the UnionInfo wizard contains a work area with one panel for each variant of the data type being developed. Since these variant panels are complicated in their own right, we have not included this code in UnionInfo but removed it to the VariantPanel class, instances of which are contained in the main dialog box.

A wizard typically cannot detect all of the errors in a design until it attempts to emit code via the produce method. In the UnionInfo class, this method simply invokes the same method on each of its VariantPanels. If one of those detects an error, it must report it by communicating the error message up to the main dialog box for display in its error panel. Since, however, VariantPanel inherits from HorizontalPanel and not from ClassUnionWizard, it must use some other mechanism to access the errorMessage method. In other words, it must *acquire* this method from its containers.

This acquisition link also makes it easier to extend UnionInfo's behavior later. In particular, if we wanted to subclass this to direct all error reports to a log file on disk, in addition to sending them to the window, we could simply place the VariantPanel objects into the extended UnionInfo; the panels would acquire the extended errorMessage method, and we would achieve the desired behavior with no changes to VariantPanel.

Without acquisition, we could achieve the same behavior through either of two strategies. First (and currently), we could wrap the errorMessage method in a Scheme closure and pass this as an argument to the VariantPanel constructor; this panel object could then simply invoke the closure to report an error. Second, we could manage the container pointer manually, and the VariantPanel object could chase pointers to find the errorMessage method.

## 3. CLASSIC JAVA

ClassicJava [7] is a formal model of the sequential core of Java [13] that includes inheritance, method overriding, and field mutation; the model could easily apply to similar OO languages, like C# [3] or Eiffel [19]. We summarize ClassicJava in this section and refer the reader to the original paper for the full details.

Figure 5 specifies the syntax for ClassicJava programs. The underlined portions of the syntax are added by the type checking and elaboration phase in order to make certain context-sensitive details available to the operational seman-

tics. For instance, evaluating a super expression requires knowledge of the location of the expression in the original program in order to perform method lookup correctly. Similarly, field accesses must know the static type of the object whose field is being accessed in order to resolve shadowed fields correctly. The underlined annotations provide all necessary information for these cases.

Figure 6 describes the six type judgments necessary to define ClassicJava's static semantics; we refer the reader to the original paper for the definition of these judgments. The ClassicJava semantics also requires several additional relations; those that are relevant to our work are defined in figure 7. The dynamic semantics (a small-step operational semantics with evaluation contexts) requires access to the program's class definitions $P$ and an explicit store $\mathcal{S}$. Finally, we represent an instance as an ordered pair $\langle c, \mathcal{F} \rangle$, where $c$ is the object's class tag and $\mathcal{F}$ is a finite map from tagged field names (i.e., $c.fd$) to values.

ClassicJava satisfies a type soundness theorem:

**Theorem 1 (Type Soundness for ClassicJava)**
*If* $\vdash_{\mathsf{p}} P \Rightarrow P' : t$ *and* $P' = defn_1 \ \ldots \ defn_n \ e$, *then either*

- $P' \vdash \langle e, \varnothing \rangle \hookrightarrow^* \langle obj, \mathcal{S} \rangle$ *and* $\mathcal{S}(obj) = \langle t', \mathcal{F} \rangle$ *and* $t' \leqslant_P t$; *or*
- $P' \vdash \langle e, \varnothing \rangle \hookrightarrow^* \langle \mathsf{null}, \mathcal{S} \rangle$; *or*
- $P' \vdash \langle e, \varnothing \rangle \hookrightarrow^* \langle \mathsf{error{:}\ bad\ cast}, \mathcal{S} \rangle$; *or*
- $P' \vdash \langle e, \varnothing \rangle \hookrightarrow^* \langle \mathsf{error{:}\ dereferenced\ null}, \mathcal{S} \rangle$; *or*
- $P' \vdash \langle e, \varnothing \rangle \Uparrow$.

# 4. JACQUES

Jacques is an extension to ClassicJava. It supports environmental acquisition of both fields and methods. In this section, we first describe Jacques's syntax, type system, and operational semantics; then we state and sketch a proof of type soundness for Jacques. In the following section, we discuss our design choices and how the formal model helped making them.

To describe object containment in Jacques programs, we use two directed graphs: the class containment graph and the object containment graph. The class containment graph for a program $P$ contains one node for each class defined in $P$; there is an edge from class $c_1$ to class $c_2$ if and only if $c_1$'s declaration indicates that it may be contained in $c_2$ (section 4.1). The class containment graph is orthogonal to the class inheritance tree, and it is a static property of program $P$.

In contrast, the object containment graph describes the state of the program at a particular point in its execution. This graph (section 4.4) contains one node for each object instance in the program's store, and an edge from object $x$ to object $y$ exactly when $x$ is contained in $y$.

Finally, while Jacques supports most of ClassicJava's capabilities, we disallow field shadowing, to simplify the description of field acquisition. In section 6, we discuss the issue of adding field shadowing and other features of Java to Jacques.

## 4.1 Syntax

To support acquisition along containment links between objects, we must have some way to distinguish between those links that represent containment in the sense of a containment hierarchy and those that do not. Therefore, in Jacques, the programmer must explicitly tag those fields that represent a containment relationship. Jacques also requires explicit acquisition, i.e., classes must explicitly declare the fields they acquire from their environment. Figure 8 extends ClassicJava's class definition syntax with these annotations.

$$
\begin{array}{lcl}
defn & = & \mathsf{class}\ c\ \mathsf{extends}\ c\ \mathsf{contained}\ c^*\ \{ \\
 & & \quad field^* \\
 & & \quad \mathsf{contains}\ field^* \\
 & & \quad \mathsf{acquires}\ field^* \\
 & & \quad meth^* \\
 & & \quad \mathsf{acquires}\ acqmeth^*\} \\
acqmeth & = & t\ md(t^*)
\end{array}
$$

**Figure 8: Jacques syntax extensions**

The contains and acquires keywords divide the field definitions into three groups. Consider the class definition

```
class A ··· {
    X x
    contains Y y
    acquires Z z ···
}
```

and let a be an instance of A. Fields in the first two groups (here, x and y) are parts of a, and fields in the last group (z) are not directly in a but are rather acquired from a's context. Further, only objects in fields in the second group (here, y) are considered to be *contained* in a, and therefore only they may acquire fields from a.

Similarly, the acquires keyword splits the method definitions into two groups. Methods in the first group behave as in ClassicJava. Methods in the second group, on the other hand, are acquired from the object's context; therefore, their definitions need not include anything beyond their signatures.

Finally, the (possibly empty) sequence of classes specified after the contained keyword places restrictions on the types of objects that may contain objects of the class being defined. In particular, if an instance $a$ of class $c$ is contained in anything, the container's type must be a subtype of one of the classes specified in $c$'s container list. As a consequence, if this list is empty, then $a$ may not be contained (in the spirit of environmental acquisition) in any object.

## 4.2 Types

Both the static and dynamic semantics must be able to distinguish between the three kinds of fields and the two kinds of methods. Therefore, we adjust ClassicJava's abstract representation of fields and methods. A field definition is now a triple $\langle t, fd, s \rangle$, where $t$ is the field's type, $fd$ is its name, and $s$ is a tag indicating the kind of field; $s$ is one of normal, contained, or acquired.

Similarly, a method definition is a 5-tuple $\langle md, T, V, e, s \rangle$, where $md$, $T$, $V$, and $e$ are as before, and $s$ is a tag indicating where the method originated; $s$ may be either normal or acquired. (The third and fourth elements of the tuple are meaningless for acquired methods, but we leave them in to simplify the definition of, e.g., $\in_P$.)

| | |
|---|---|
| $\vdash_{\mathsf{p}} P \Rightarrow P' : t$ | program $P$ elaborates to $P'$ with type $t$ |
| $P \vdash_{\mathsf{d}} defn \Rightarrow defn'$ | class definition $defn$ elaborates to $defn'$ |
| $P, t \vdash_{\mathsf{m}} meth \Rightarrow meth'$ | method $meth$ in class $t$ elaborates to $meth'$ |
| $P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t$ | $e$ elaborates to $e'$ with type $t$ in $\Gamma$ |
| $P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t$ | $e$ elaborates to $e'$ and has type $t$ using subsumption in $\Gamma$ |
| $P \vdash_{\mathsf{t}} t$ | type $t$ exists |

**Figure 6:** ClassicJava **type judgments**

$\prec_P$ immediate subclass
$c \prec_P c' \iff \mathsf{class}\ c\ \mathsf{extends}\ c'\ \{\cdots\}$ is in $P$

$\leqslant_P$ subclass
$\leqslant_P$ is defined to be the reflexive-transitive closure of $\prec_P$

$\notin_P$ field is declared in a class
$\langle c.fd, t\rangle \notin_P c \iff \mathsf{class}\ c\ \cdots\ \{\cdots\ t\ fd\ \cdots\}$ is in $P$

$\notin_P$ method is declared in a class
$\langle md, (t_1 \ldots t_n \to t), (var_1 \ldots var_n), e\rangle \notin_P c \iff \mathsf{class}\ c\ \cdots\ \{\cdots\ t\ md(t_1\ var_1\ \ldots\ t_n\ var_n)\ \{e\}\ \cdots\}$ is in $P$

$\in_P$ field is contained in a class
$\langle c'.fd, t\rangle \in_P c \iff \langle c'.fd, t\rangle \notin_P c'$ and $c' = \min\{c'' \mid c \leqslant_P c''$ and $\exists t' . \langle c''.fd, t'\rangle \notin_P c''\}$

$\in_P$ method is contained in a class
$\langle md, T, V, e\rangle \in_P c \iff \langle md, T, V, e\rangle \notin_P c'$ and $c' = \min\{c'' \mid c \leqslant_P c''$ and $\exists e', V' . \langle md, T, V', e'\rangle \in_P c''\}$

**Figure 7:** ClassicJava **relations**

Finally, we also use an adjusted instance representation: $\langle c, ctnr, \mathcal{F}\rangle$ where $c$ is the class tag, $ctnr$ is a (possibly null) reference to the instance's container, and $\mathcal{F}$ is a finite map from field names to values. We also define three selector functions: *classTag*, *ctnr*, and *fields* map an instance triple to its class tag, container reference, and field function, respectively.

Jacques's semantics requires many of the relations and predicates defined in figure 7, with appropriate modifications to several, including $\notin_P$ and $\in_P$. The additional necessary predicates are straightforward; their definitions are provided below and in figure 9.

The predicate $canAcq_P(c, \langle t, fd, \mathsf{acquired}\rangle)$ determines if it is safe for class $c$ to acquire field $\langle t, fd, s\rangle$. This acquisition is safe if the class containment graph implies that, for all possible environments for instances of $c$, the nearest enclosing class that provides the field $fd$ provides it with a compatible type. Formally, we define $canAcq_P(c, \langle t, fd, s\rangle)$ to be true if and only if every unsafe path that starts with $c$ in the class containment graph has a safe prefix. An unsafe path is one that ends in a class $c''$ such that

- $\langle t'', fd, s''\rangle \in_P c''$, and
- $t'' \not\leqslant_P t$.

A safe path is one that ends in a class $c'$ such that

- $\langle t', fd, s'\rangle \in_P c'$, and
- $s' \neq \mathsf{acquired}$, and
- $t' \leqslant_P t$.

In other words, no traversal of the class containment graph finds a field of the desired name and an incompatible type before it finds a field with a compatible type. Note that we allow the existence of paths that never find a field of the desired name, and we even allow the absence of any paths that find the desired field. Forbidding these conditions would not strengthen the type system: even if all paths from a class $c$ reach the field, we must still check at run-time for the possibility that an object of type $c$ has a null container.

Similarly, the predicate $canAcqMeth_P$ checks whether it is safe for class $c$ to acquire a method $md$; it differs only in the type compatibility requirements. Formally, $canAcqMeth_P(c, \langle md, (t_1 \ldots t_n \to t), V, e, s\rangle)$ is true if and only if every unsafe path that starts with $c$ in the class containment graph has a safe prefix. Here, an unsafe path is one that ends in a class $c''$ such that $\langle md, (t_1'' \ldots t_m'' \to t''), V'', e'', s''\rangle \in_P c''$ and one of the following is true:

- $m \neq n$, or
- $t'' \not\leqslant_P t$, or
- $m = n$ and for some $i \in [1, n]$, $t_i \not\leqslant_P t_i''$.

A safe path is one that ends in a class $c'$ where

- $\langle md, (t_1' \ldots t_n' \to t'), V', e', s'\rangle \in_P c'$, and
- $s' \neq \mathsf{acquired}$, and
- $t' \leqslant_P t$, and
- $t_i \leqslant_P t_i'$ for all $i \in [1, n]$.

With these relations on the class graph, we can define the type elaboration rules for Jacques; figure 10 specifies the necessary changes to ClassicJava's rules. In addition to the rules shown, there are straightforward variations of ClassicJava's Get, Call, and Super to account for the change in field and method representations. Also, because we no longer support field shadowing, we add $ClassFieldsOK(P)$ as an antecedent to the $\textsc{Prog}_j$ rule, and neither the $\textsc{Get}_j$

$ClassFieldsOK(P)$ no field definition shadows a field defined in a superclass
$(\langle t_1, fd, s_1\rangle \notin_P c_1$ and $\langle t_2, fd, s_2\rangle \notin_P c_2$ and $c_1 \neq c_2) \implies c_1 \not\leqslant_P c_2$

$containers_P(c)$ set of all possible containers of instances of class $c$
$containers_P(\mathsf{Object}) = \varnothing$
$containers_P(c) = \{c_1, \ldots, c_n\}$ where $\mathsf{class}\ c\ \cdots\ \mathsf{contained}\ c_1 \ldots c_n\ \{\ \cdots\ \}$ is in $P$

$acqFields_P(c)$ set of all acquired fields in class $c$
$acqFields_P(\mathsf{Object}) = \varnothing$
$acqFields_P(c) = acqFields_P(c') \cup \{\langle t, fd, \mathsf{acquired}\rangle \mid \langle t, fd, \mathsf{acquired}\rangle \notin_P c\}$
where $\mathsf{class}\ c\ \mathsf{extends}\ c'\ \cdots\ \{\ \cdots\ \}$ is in $P$

$acqMethods_P(c)$ set of all acquired methods in class $c$
$acqMethods_P(\mathsf{Object}) = \varnothing$
$acqMethods_P(c) = \{\langle md, T, V, e, \mathsf{acquired}\rangle \mid \langle md, T, V, e, \mathsf{acquired}\rangle \in_P c\}$
where $\mathsf{class}\ c\ \cdots\ \{\ \cdots\ \}$ is in $P$

$\sqsubset_P$ class may be contained
$c_1 \sqsubset_P c_2 \iff$ there exists $c' \in containers_P(c_1)$ such that $c_2 \leqslant_P c'$

**Figure 9:** JACQUES **relations**

$\textsc{Defn}_j$

$$\frac{\begin{array}{c}P \vdash_{\mathsf{t}} t_j \text{ for all } j \in [1,m] \qquad P \vdash_{\mathsf{t}} c_j \text{ for all } j \in [1,n] \qquad P, c \vdash_{\mathsf{m}} meth_i \Rrightarrow meth'_i \text{ for all } i \in [1,p] \\ \forall f \in acqFields_P(c)\,.\,canAcq_P(c,f) \qquad \forall m \in acqMethods_P(c)\,.\,canAcqMeth_P(c,m) \\ containers_P(c') \subseteq containers_P(c) \qquad t_i \sqsubset_P c \text{ for all } i \in [k+1,\ell]\end{array}}{\begin{array}{rcl}P \vdash_{\mathsf{d}} \mathsf{class}\ c\ \mathsf{extends}\ c'\ \mathsf{contained}\ c_1,\ldots,c_n & \Rrightarrow & \mathsf{class}\ c\ \mathsf{extends}\ c'\ \mathsf{contained}\ c_1,\ldots,c_n \\ \{t_1\ fd_1 \ldots t_k\ fd_k & & \{t_1\ fd_1 \ldots t_k\ fd_k \\ \mathsf{contains}\ t_{k+1}\ fd_{k+1} \ldots t_\ell\ fd_\ell & & \mathsf{contains}\ t_{k+1}\ fd_{k+1} \ldots t_\ell\ fd_\ell \\ \mathsf{acquires}\ t_{\ell+1}\ fd_{\ell+1} \ldots t_m\ fd_m & & \mathsf{acquires}\ t_{\ell+1}\ fd_{\ell+1} \ldots t_m\ fd_m \\ meth_1 \ldots meth_p & & meth'_1 \ldots meth'_p \\ \mathsf{acquires}\ acqmeth_1 \ldots acqmeth_q\} & & \mathsf{acquires}\ acqmeth_1 \ldots acqmeth_q\}\end{array}}$$

$\textsc{Set}_j$

$$\frac{P, \Gamma \vdash_{\mathsf{s}} e_1 \Rrightarrow e'_1\ :\ t' \qquad \langle t, fd, s\rangle \in_P t' \qquad s \neq \mathsf{acquired} \qquad P, \Gamma \vdash_{\mathsf{s}} e_2 \Rrightarrow e_2\ :\ t}{P, \Gamma \vdash_{\mathsf{e}} e_1.fd = e_2 \Rrightarrow e'_1.fd = e'_2\ :\ t}$$

**Figure 10:** JACQUES **type elaboration rules**

nor the $\textsc{Set}_j$ rule annotates field reference or assignment expressions. Further, $\textsc{Set}_j$ prohibits assignments to acquired fields; see section 5.5 for the rationale for this restriction.

The $\textsc{Defn}_j$ rule contains most of the changes that support acquisition. First, it verifies that class $c$'s environments provide the necessary fields and methods with compatible types. This indirectly requires that if $c$ or any of its superclasses acquire any fields or methods, then $c$ must have at least one container. Second, $c$'s containers must be a superset of $c$'s superclass's containers, to preserve behavioral subtyping.[2] Third, it must ensure that all of the classes that can be in $c$'s contained fields allow themselves to be contained in $c$, as required by the soundness proof. The $\textsc{Defn}_j$ rule could also verify that class $A$ can contain instances of class $B$ if and only if $B$ can be contained in $A$. While warnings of violations of this invariant are likely to be useful to the programmer, they are not required for type soundness.

[2]As we move down the inheritance hierarchy, the set of possible containers must increase monotonically. However, a subclass may also add new acquired fields or methods; this further constrains the set of possible containers. There is no inconsistency hidden here; it simply creates a trade-off that the programmer must consider during the design of class hierarchies.

## 4.3 Semantics

JACQUES's semantics is like that of CLASSICJAVA, with some modifications and additions; the changes are defined in figure 11, with additional supplementary functions defined in figure 12. In particular, many of these rules perform the additional run-time checks necessary to ensure that acquisition is well-defined. In particular, the $aget_j$, $acall_j$, and $asuper_j$ rules ensure that references to acquired fields or methods happen when the object is in a sufficiently rich environment; if not, the rules raise an "incomplete context" exception. Similarly, the $ct\text{-}set_j$ rule, for assignment to a contained field, ensures that the field's new contents are not already contained elsewhere and that the assignment does not create an acquisition cycle. If either condition fails, the rules raise an "already contained" or "container cycle" exception as appropriate.

## 4.4 Jacques Soundness

To study the properties of JACQUES's type system and semantics, we have implemented the model in PLT Redex [18], a system for debugging reduction systems. This implementation does not yet support either method acquisition or field assignment, but it raises our confidence that the proof of the following type-soundness theorem is correct.

$$
\begin{array}{rcl}
e & = & \ldots \mid obj \\
v & = & obj \mid \mathsf{null} \\
\mathsf{E} & = & [\,] \mid \mathsf{E}.fd \mid \mathsf{E}.fd = e \mid v.fd = \mathsf{E} \mid \mathsf{E}.md(e\ldots) \mid v.md(v\ldots\mathsf{E}\ e\ldots) \\
 & & \mid \mathsf{super}\ \underline{\equiv v : c}.md(v\ldots\mathsf{E}\ e\ldots) \mid \mathsf{cast}\ t\ \mathsf{E} \mid \mathsf{let}\ var = \mathsf{E}\ \mathsf{in}\ e
\end{array}
$$

$P \vdash \langle \mathsf{E}[\mathsf{new}\ c], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[obj], \mathcal{S}[obj \mapsto \langle c, \mathsf{null}, \mathcal{F}\rangle]\rangle$ $[new_j]$
where $obj \notin \mathrm{dom}(\mathcal{S})$ and $\mathcal{F} = \{fd \mapsto \mathsf{null} \mid \langle t, fd, s\rangle \in_P c$ and $s \neq \mathsf{acquired}\}$

$P \vdash \langle \mathsf{E}[obj.fd], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathcal{F}(fd)], \mathcal{S}\rangle$ $[get_j]$
where $\mathcal{F} = \mathit{fields}(\mathcal{S}(obj))$ and $fd \in \mathrm{dom}(\mathcal{F})$

$P \vdash \langle \mathsf{E}[obj.fd], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathit{getAcqField}_P(\mathcal{S}, obj, fd)], \mathcal{S}\rangle$ $[aget_j]$
where $\langle t, fd, \mathsf{acquired}\rangle \in_P \mathit{classTag}(\mathcal{S}(obj))$
and $\mathit{getAcqField}_P(\mathcal{S}, obj, fd) \neq \bot$

$P \vdash \langle \mathsf{E}[obj.fd], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error}\text{: incomplete context}, \mathcal{S}\rangle$ $[xaget_j]$
where $\langle t, fd, \mathsf{acquired}\rangle \in_P \mathit{classTag}(\mathcal{S}(obj))$
and $\mathit{getAcqField}_P(\mathcal{S}, obj, fd) = \bot$

$P \vdash \langle \mathsf{E}[obj.fd = v], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathit{updateField}(\mathcal{S}, obj, fd, v)\rangle$ $[set_j]$
where $\langle t, fd, \mathsf{normal}\rangle \in_P \mathit{classTag}(\mathcal{S}(obj))$

$P \vdash \langle \mathsf{E}[obj.fd = \mathsf{null}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathsf{null}], \mathit{updateCtdField}(\mathcal{S}, obj, fd, \mathsf{null})\rangle$ $[n\text{-}ct\text{-}set_j]$
where $\langle t, fd, \mathsf{contained}\rangle \in_P \mathit{classTag}(\mathcal{S}(obj))$

$P \vdash \langle \mathsf{E}[obj.fd = v], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error}\text{: already contained}, \mathcal{S}\rangle$ $[x\text{-}ct\text{-}set_j]$
where $\langle t, fd, \mathsf{contained}\rangle \in_P \mathit{classTag}(\mathcal{S}(obj))$
and $v \in \mathrm{dom}(\mathcal{S})$ and $\mathit{ctnr}(\mathcal{S}(v)) \neq \mathsf{null}$

$P \vdash \langle \mathsf{E}[obj.fd = v], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error}\text{: container cycle}, \mathcal{S}\rangle$ $[cycle\text{-}set_j]$
where $\langle t, fd, \mathsf{contained}\rangle \in_P \mathit{classTag}(\mathcal{S}(obj))$
and $v \in \mathrm{dom}(\mathcal{S})$ and $\mathit{canReach}_P(\mathcal{S}, obj, v)$

$P \vdash \langle \mathsf{E}[obj.fd = v], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathit{updateCtdField}(\mathcal{S}, obj, fd, v)\rangle$ $[ct\text{-}set_j]$
where $v \in \mathrm{dom}(\mathcal{S})$ and $\langle t, fd, \mathsf{contained}\rangle \in_P \mathit{classTag}(\mathcal{S}(obj))$
and $\mathit{ctnr}(\mathcal{S}(v)) = \mathsf{null}$ and $\neg\, \mathit{canReach}_P(\mathcal{S}, obj, v)$

$P \vdash \langle \mathsf{E}[obj.md(v_1\ \ldots\ v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e_b[obj'/\mathsf{this}, v_1/var_1, \ldots, v_n/var_n]], \mathcal{S}\rangle$ $[acall_j]$
where $\langle md, T, V, e, \mathsf{acquired}\rangle \in_P \mathit{classTag}(\mathcal{S}(obj))$
and $\langle obj', e_b\rangle = \mathit{getAcqMethod}_P(\mathcal{S}, obj, md)$

$P \vdash \langle \mathsf{E}[obj.md(v_1\ \ldots\ v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error}\text{: incomplete context}, \mathcal{S}\rangle$ $[xacall_j]$
where $\langle md, T, V, e, \mathsf{acquired}\rangle \in_P \mathit{classTag}(\mathcal{S}(obj))$
and $\mathit{getAcqMethod}_P(\mathcal{S}, obj, md) = \bot$

$P \vdash \langle \mathsf{E}[\mathsf{super}\ \underline{\equiv obj : c'}.md(v_1\ \ldots\ v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[obj/\mathsf{this}, v_1/var_1, \ldots, v_n/var_n]], \mathcal{S}\rangle$ $[super_j]$
where $\overline{\langle md, T, (var_1, \ldots, var_n), e, \mathsf{normal}\rangle} \in_P c'$

$P \vdash \langle \mathsf{E}[\mathsf{super}\ \underline{\equiv obj : c'}.md(v_1\ \ldots\ v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[obj'/\mathsf{this}, v_1/var_1, \ldots, v_n/var_n]], \mathcal{S}\rangle$ $[asuper_j]$
where $\overline{\langle md, T, \_, \_, \mathsf{acquired}\rangle} \in_P c'$ and $\langle obj', e\rangle = \mathit{getAcqMethod}_P(\mathcal{S}, obj, md)$

$P \vdash \langle \mathsf{E}[\mathsf{super}\ \underline{\equiv obj : c'}.md(v_1\ \ldots\ v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error}\text{: incomplete context}, \mathcal{S}\rangle$ $[xasuper_j]$
where $\overline{\langle md, T, \_, \_, \mathsf{acquired}\rangle} \in_P c'$ and $\mathit{getAcqMethod}_P(\mathcal{S}, obj, md) = \bot$

$P \vdash \langle \mathsf{E}[\mathsf{cast}\ t'\ obj], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[obj], \mathcal{S}\rangle$ $[cast]$
where $\mathcal{S}(obj) = \langle c, ctnr, \mathcal{F}\rangle$ and $c \leqslant_P t'$

$P \vdash \langle \mathsf{E}[\mathsf{let}\ var = v\ \mathsf{in}\ e], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[v/var]], \mathcal{S}\rangle$ $[let]$

$P \vdash \langle \mathsf{E}[\mathsf{cast}\ t'\ obj], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error}\text{: bad cast}, \mathcal{S}\rangle$ $[xcast]$
where $\mathcal{S}(obj) = \langle c, ctnr, \mathcal{F}\rangle$ and $c \not\leqslant_P t'$

$P \vdash \langle \mathsf{E}[\mathsf{cast}\ t'\ \mathsf{null}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error}\text{: dereferenced null}, \mathcal{S}\rangle$ $[ncast]$

$P \vdash \langle \mathsf{E}[\mathsf{null}.fd], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error}\text{: dereferenced null}, \mathcal{S}\rangle$ $[nget]$

$P \vdash \langle \mathsf{E}[\mathsf{null}.fd = v], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error}\text{: dereferenced null}, \mathcal{S}\rangle$ $[nset]$

$P \vdash \langle \mathsf{E}[\mathsf{null}.md(v_1\ \ldots\ v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error}\text{: dereferenced null}, \mathcal{S}\rangle$ $[ncall]$

**Figure 11: JACQUES operational semantics**

$$
\begin{array}{rcl}
getAcqField_P(\mathcal{S}, \mathsf{null}, fd) & = & \bot \\
getAcqField_P(\mathcal{S}, obj, fd) & = & \left\{ \begin{array}{ll} \mathcal{F}(fd) & \text{if } fd \in \mathrm{dom}(\mathcal{F}) \\ getAcqField_P(\mathcal{S}, ctnr, fd) & \text{otherwise} \end{array} \right. \\
 & & \quad \text{where } \langle c, ctnr, \mathcal{F} \rangle = \mathcal{S}(obj) \\
 & & \\
getAcqMethod_P(\mathcal{S}, \mathsf{null}, fd) & = & \bot \\
getAcqMethod_P(\mathcal{S}, obj, fd) & = & \left\{ \begin{array}{ll} \langle obj, e_b \rangle & \text{if } \langle md, T, V, e_b, \mathsf{normal} \rangle \in_P classTag(\mathcal{S}(obj)) \\ getAcqMethod_P(\mathcal{S}, ctnr, md) & \text{otherwise} \end{array} \right. \\
 & & \quad \text{where } ctnr = ctnr(\mathcal{S}(obj)) \\
 & & \\
updateField(\mathcal{S}, obj, fd, v) & = & \mathcal{S}[obj \mapsto \langle c, ctnr, \mathcal{F}[fd \mapsto v] \rangle] \\
 & & \quad \text{where } \langle c, ctnr, \mathcal{F} \rangle = \mathcal{S}(obj) \\
 & & \\
updateCtdField(\mathcal{S}, obj, fd, v) & = & updateField(\mathcal{S}_2, obj, fd, v) \\
 & & \quad \text{where } \langle c, ctnr, \mathcal{F} \rangle = \mathcal{S}(obj) \text{ and} \\
 & & \qquad \mathcal{S}_1 = updateContainer(\mathcal{S}, \mathcal{F}(fd), \mathsf{null}) \text{ and} \\
 & & \qquad \mathcal{S}_2 = updateContainer(\mathcal{S}_1, v, obj) \\
 & & \\
updateContainer(\mathcal{S}, obj, ctnr) & = & \left\{ \begin{array}{ll} \mathcal{S} & \text{if } obj = \mathsf{null} \\ \mathcal{S}[obj \mapsto \langle c, ctnr, \mathcal{F} \rangle] & \text{otherwise} \\ \quad \text{where } \langle c, \_, \mathcal{F} \rangle = \mathcal{S}(obj) & \end{array} \right. \\
 & & \\
canReach_P(\mathcal{S}, obj, v) & \iff & obj = v \text{ or } (ctnr \neq \mathsf{null} \text{ and } canReach_P(\mathcal{S}, ctnr, v)) \\
 & & \quad \text{where } ctnr = ctnr(\mathcal{S}(obj))
\end{array}
$$

**Figure 12:** JACQUES **supplementary functions**

**Theorem 2 (Type Soundness for** JACQUES**)**
*If* $\vdash_{\mathsf{p}} P \Rightarrow P' : t$ *and* $P' = defn_1 \ \ldots \ defn_n \ e$*, then either*

- $P' \vdash \langle e, \varnothing \rangle \Uparrow$*; or*
- $P' \vdash \langle e, \varnothing \rangle \hookrightarrow^* \langle obj, \mathcal{S} \rangle$ *and* $classTag(\mathcal{S}(obj)) \leqslant_P t$*; or*
- $P' \vdash \langle e, \varnothing \rangle \hookrightarrow^* \langle \mathsf{null}, \mathcal{S} \rangle$*; or*
- $P' \vdash \langle e, \varnothing \rangle \hookrightarrow^* \langle \mathsf{error{:}\ bad\ cast}, \mathcal{S} \rangle$*; or*
- $P' \vdash \langle e, \varnothing \rangle \hookrightarrow^* \langle \mathsf{error{:}\ dereferenced\ null}, \mathcal{S} \rangle$*;*

*or*

- $P' \vdash \langle e, \varnothing \rangle \hookrightarrow^* \langle \mathsf{error{:}\ incomplete\ context}, \mathcal{S} \rangle$*; or*
- $P' \vdash \langle e, \varnothing \rangle \hookrightarrow^* \langle \mathsf{error{:}\ already\ contained}, \mathcal{S} \rangle$*; or*
- $P' \vdash \langle e, \varnothing \rangle \hookrightarrow^* \langle \mathsf{error{:}\ container\ cycle}, \mathcal{S} \rangle$.

A comparison of theorem 1 and theorem 2 indicates that JACQUES can throw additional exceptions that do not occur in CLASSICJAVA programs. The last three exceptions are forced upon us by acquisition, and we cannot avoid them without a complex analysis, a more sophisticated type system, or different linguistic mechanisms. Put differently, we have succeeded in making acquisition just as safe as ML, as many errors can be caught statically, but some run-time checks (and exceptions) remain necessary.

The soundness proof follows the standard method [23] as applied to CLASSICJAVA [7]. In addition to the subject reduction and uniform evaluation lemmas, we also require some definitions and supporting lemmas beyond those used in the CLASSICJAVA soundness proof; they are stated below.

**Definition 1 (Object Containment)**
We write $P, \mathcal{S} \vdash obj_1 \sqsupseteq obj_2$ to indicate that $obj_1$ contains $obj_2$ in store $\mathcal{S}$ and program $P$. Formally:

$$
\begin{array}{l}
P, \mathcal{S} \vdash obj_1 \sqsupseteq obj_2 \iff \\
\quad \mathcal{S}(obj_1) = \langle c, ctnr, \mathcal{F} \rangle \text{ and } \mathcal{F}(fd) = obj_2 \\
\quad \text{for some } fd \text{ such that } \langle c', fd, \mathsf{contained} \rangle \in_P c.
\end{array}
$$

**Definition 2 (Object Containment Graph)**
For a given store $\mathcal{S}$, we construct a directed graph $G_{\mathcal{S}}$, called the object containment graph, as follows:

$$G_{\mathcal{S}} = (V, E)$$

where

$$V = \mathrm{rng}(\mathcal{S}) \text{ and } E = \{\langle x, y \rangle \in V^2 \mid ctnr(x) = y\}.$$

In other words, there is a node in $G_{\mathcal{S}}$ for each object in $\mathcal{S}$, and an edge from each node to its container.

Next we need to adapt the environment-store consistency relation from the proof of theorem 1 to the new context.

**Definition 3 (Environment-Store Consistency)**
We write $P, \Gamma \vdash_\sigma \mathcal{S}$ to indicate that the type environment $\Gamma$ and the store $\mathcal{S}$ are consistent with one another, given the program $P$. Formally:

$$
\begin{array}{lll}
 & P, \Gamma \vdash_\sigma \mathcal{S} \iff & \\
 & \quad obj \in \mathrm{dom}(\Gamma) \implies obj \in \mathrm{dom}(\mathcal{S}) & \\
 & \wedge \quad \mathrm{dom}(\mathcal{S}) \subseteq \mathrm{dom}(\Gamma) & \\
* & \wedge \quad G_{\mathcal{S}} \text{ has no cycles and is of finite size} & \\
 & \wedge \quad [\mathcal{S}(obj) = \langle c, ctnr, \mathcal{F} \rangle \implies & \\
 & \qquad \Gamma(obj) = c & \\
 & \qquad \wedge \quad \mathrm{dom}(\mathcal{F}) = \{fd \mid \langle t, fd, s \rangle \in_P c, s \in \{\mathsf{normal}, \mathsf{contained}\}\} & \\
 & \qquad \wedge \quad \mathrm{rng}(\mathcal{F}) \subseteq \mathrm{dom}(\mathcal{S}) \cup \{\mathsf{null}\} & \\
 & \qquad \wedge \quad (\mathcal{F}(fd) \in \mathrm{dom}(\mathcal{S}) \wedge \langle c', fd, s \rangle \in_P c) \implies & \\
 & \qquad\quad classTag(\mathcal{S}(\mathcal{F}(fd))) \leqslant_P c' & \\
* & \qquad \wedge \quad (\mathcal{F}(fd) = \mathcal{F}(fd') \neq \mathsf{null} \wedge \langle t, fd, \mathsf{contained} \rangle \in_P c \wedge & \\
 & \qquad\quad \langle t', fd', \mathsf{contained} \rangle \in_P c) \implies fd = fd' & \\
* & \qquad \wedge \quad ctnr = \mathsf{null} \implies \neg \exists obj' \in \mathrm{dom}(\mathcal{S}) \,.\, P, \mathcal{S} \vdash obj' \sqsupseteq obj & \\
* & \qquad \wedge \quad ctnr \in \mathrm{dom}(\mathcal{S}) \implies P, \mathcal{S} \vdash ctnr \sqsupseteq obj & \\
* & \qquad \wedge \quad ctnr \in \mathrm{dom}(\mathcal{S}) \implies & \\
 & \qquad\quad \exists c' \in containers_P(c) \,.\, classTag(\mathcal{S}(ctnr)) \leqslant_P c' & \\
* & \qquad \wedge \quad (P, \mathcal{S} \vdash obj \sqsupseteq obj') \implies ctnr(\mathcal{S}(obj')) = obj & \\
* & \qquad \wedge \quad ctnr \in \mathrm{dom}(\mathcal{S}) \cup \{\mathsf{null}\}] &
\end{array}
$$

Those clauses marked with $*$ are added to ClassicJava for Jacques. In order, they place the following requirements on the store:

- No object may have two contained fields that refer to the same object. This allows us to set an object's container to null when we remove it from its container without breaking any other invariants.
- If *obj*'s container is null, then there exists no object that contains *obj*.
- If *obj*'s container is *obj'*, then *obj'* contains *obj*.
- If *obj* has class tag $c$ and container *obj'*, then *obj'* must have a class tag that is a subtype of one of class $c$'s containers.
- If *obj* contains *obj'*, then *obj'* must specify *obj* as its container.
- An object's container is either a valid object reference or null.

**Lemma 1 (Containment Path Consistency)** *If* $P, \Gamma \vdash_\sigma \mathcal{S}$ *and* $\mathcal{S}(obj) = \langle c, ctnr, \mathcal{F} \rangle$ *and* $ctnr \neq$ null, *then there exists a class* $c' \in containers_P(c)$ *such that*

$$classTag(\mathcal{S}(ctnr)) \leqslant_P c'.$$

This lemma follows fairly directly from $P, \Gamma \vdash_\sigma \mathcal{S}$. Since $G_\mathcal{S}$ is finite and acyclic, all paths have finite length, and therefore a simple induction on the length of the path proves that all paths in the object containment graph are consistent with the class containment graph. We use this result in the proof of the subject reduction lemma to ensure that the results of $getAcqField_P$ and $getAcqMethod_P$ have the expected types.

# 5. JACQUES DESIGN ISSUES

In developing Jacques, we encountered several interesting design problems. We rejected some design alternatives because they break type safety; for others, we based our decisions on pragmatics. In this section, we discuss these design choices and provide rationales. We use the program in figure 13 as a running example.

## 5.1 Containment Cycles

The object containment graph cannot have cycles, as Gil and Lorenz explain. This does not mean, however, that the *class* containment graph should be acyclic as well. As stated in section 2, acquisition is often useful in conjunction with the composite pattern. It is frequently essential in this pattern that objects be allowed to contain other instances of their own class (or a superclass). For a concrete example, consider a GUI toolkit library in which classes such as Panel and Button are subclasses of Component. If instances of Panel cannot contain other instances of Panel or instances of (subclasses of) Component, then programmers have too little freedom to compose objects. Therefore, we allow cycles within the class containment graph. Acquisition remains well-defined as long as the *object* containment graph has no cycles, and the *ct-set*$_j$ reduction enforces this restriction at runtime.

This design choice is orthogonal to type safety. As discussed in section 4.4, we have proved type soundness with cycles in the class containment graph.

## 5.2 Acquisition by Value and by Name

In their original description of acquisition, Gil and Lorenz do not address field assignment and how it interacts with acquisition. With assignment, it becomes important to determine when acquisition takes place. We considered two options, which we call "acquisition-by-value" and "acquisition-by-name," by analogy with call-by-value and call-by-name. The primary difference between the two is the point at which the acquiring object receives the value of the container's field.

In our running example (figure 13) with acquisition-by-value, the acquiring object anItem receives the value of its acquired fields when it is placed in its container aCtnr. Subsequent modifications to aCtnr's fields are not automatically visible to anItem. With acquisition-by-name, on the other hand, anItem receives the value of its acquired fields anew each time the fields are referenced. Therefore, modifications to aCtnr's fields are automatically visible to anItem. In both cases, type soundness is preserved as long as the value in aCtnr.fd has a type compatible with that declared for anItem.fd.

The choice of acquisition-by-value raises two questions:

1. If we remove anItem from its container aCtnr, does it preserve the field values it acquired from aCtnr, or do those fields now become undefined?
2. If we then add anItem to another container aCtnr', does it preserve its existing field values, or does it replace them with the values then current in its new container?

With acquisition-by-name, in contrast, the answers to these questions follow directly from the definition above. This ambiguity leads us to believe that acquisition-by-name is the better mechanism, from both a semantic as well as a pragmatic perspective.

## 5.3 Types of Acquired Fields

In the example of figure 13, anItem acquires the field fd from its environment, specifically the container aCtnr. The class Container1 defines fd to have type Prop1, but Item acquires it with Property's interface. Since Prop1 is a subtype of Property, this acquisition is safe. In general, acquiring classes may expect a more general type for their acquired fields than their environments actually provide. For increased flexibility, the definition of Jacques, through the $canAcq_P$ predicate, allows just this sort of type variance. (Note that this is similar to, though not exactly the same as, the traditional notion of contravariance: we have Prop1 $\leqslant_P$ Property but Item $\not\leqslant_P$ Container1, so the idea of one type getting larger while the other gets smaller is not applicable.)

## 5.4 Types of Acquired Methods

In our example, Item acquires the method meth with argument type Prop2 and result type Property. This is safe so long as the method that is actually executed as as result of calls to anItem.meth (that is, the method provided by the context) allows a more general type for its arguments and returns a value of a more specific type. So, since Prop2 $\leqslant_P$ Property and Prop1 $\leqslant_P$ Property, Item can safely acquire meth from Container1. The definition of $canAcqMeth_P$ allows this flexibility. This is only safe, however, because Jacques does not support method overloading. As we scale acquisi-

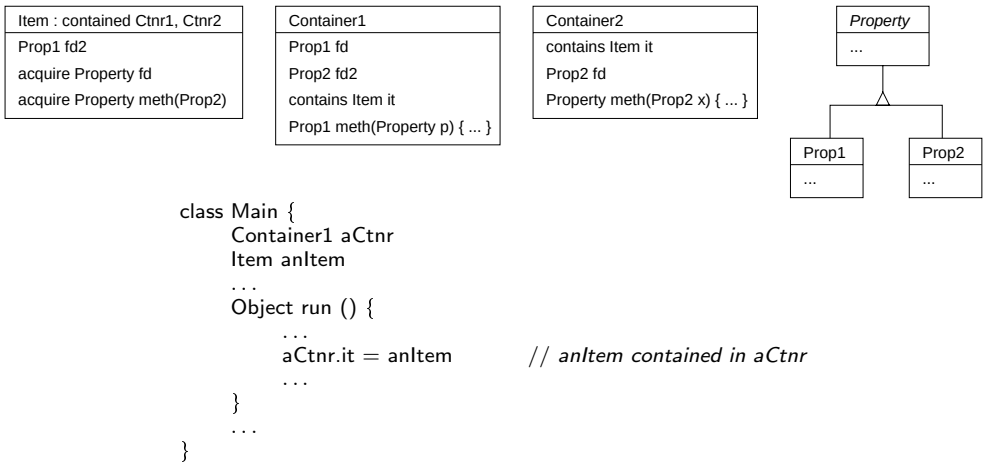



**Figure 13:** JACQUES **running example**

tion to a language with method overloading, we may have to restrict method acquisition to use invariant types.

## 5.5 Assignment to Acquired Fields

Allowing assignment to acquired fields with "contravariant" typing as described above is not type-safe. In our running example, an assignment such as

anItem.fd = new Prop2()

would type-check because anItem.fd is declared to have type Property. However, since anItem acquires the field fd from aCtnr, and aCtnr.fd has type Prop1, allowing assignments such as the above would break type safety. There are three possible strategies that would allow assignment to anItem.fd while preserving type safety:

1. The type of an acquired field must match exactly in the acquiring class and the container. In our example, this would correspond to requiring Item to acquire the field fd with type Prop1 (which would in turn prevent Container2 from containing Item).

2. We could restrict the assignment statement while preserving variance in acquired field types. In this case, the value on the right-hand side of an assignment must have exactly the same type as the left-hand side; subtyping would no longer be allowed here.

3. We could preserve the variance in acquired field types and simply disallow all assignments to acquired fields.

The second option has several drawbacks. First, as long as we preserve subsumption throughout the rest of the language, this restriction could only be performed by a dynamic check, which would complicate the semantics yet further. Second, it would restrict the types allowed on the left-hand side of an assignment statement. In Java, for instance, the left-hand side of such an assignment statement could not have an interface type. Because interfaces can never be instantiated directly, it would be impossible for the value of the right-hand side to have exactly the same type as the left. This would severely constrain assignment statements. Finally, it would forbid subsumption in a particular context. Not only would this create asymmetry in the language design, it would do so in a particularly restrictive fashion, due to the frequent occurrence of assignment statements in object-oriented programs.

Therefore, we must choose between the first and third options. JACQUES implements the third option, as we conjecture that it is more pragmatically useful than the first. Without the benefit of experience with acquisition in real-world software projects, though, we cannot answer this question definitively, and we need to revisit this issue once we have gained more experience.

## 5.6 Changing Containers

With the ability to mutate fields in existing objects comes the ability to mutate the object containment graph after it has been established. While this ability may be necessary in some applications, we want to preserve the invariant that aCtnr contains anItem if and only if anItem's container is aCtnr. This allows both objects, but particularly aCtnr, to make useful assumptions about the object containment graph at runtime.

For this reason, we prohibit programs from moving an object from one container to another in a single assignment statement, because it can easily break the containment hierarchy property. This restriction is enforced by the $x\text{-}ct\text{-}set_j$ reduction, which requires an object's container pointer to be null before it can be added to a container. An alternative would be to follow Alan Kay's maxim of replacing assignment with higher-level forms of mutation operations [14], specifically a switch-container operation.

## 5.7 Forwarding and Delegation Semantics

Gil and Lorenz discuss the choice between forwarding and delegation semantics for acquired methods. To define these terms as applied to environmental acquisition, consider our running example, in which anItem acquires the method meth from aCtnr; recall that Item $\not\leq_P$ Container1. Under forwarding semantics, when Main invokes anItem.meth, this is bound to aCtnr during the execution of the method's body. With delegation semantics, this is instead bound to anItem.

By type rule METH$_j$, the body of Container1.meth is elaborated under the assumption that this has type Container1 (or any subtype). Hence, the body can refer to arbitrary features of Container1, such as fd2. If, instead, this were suddenly bound to anItem at runtime, references to properties of Container1 may become undefined or ill-typed. Specifically, a reference to this.fd2 would reduce to a value of type Prop1 rather than Prop2 as expected. We thus conclude that forwarding semantics for environmental acquisition is the only safe possibility in a statically typed language.

## 5.8 Guarded Field References

As defined, JACQUES throws an exception upon a reference to an acquired field if the object in question is not in an environment that provides the expected field. In their original formulation, Gil and Lorenz propose guarded field references as a protection against this exception. A guarded field reference is simply a field reference expression that includes a default value to be used if the object is not in a suitable environment. If the compiler can prove that, for a particular field reference, the object is always in a suitable environment, the programmer may omit the default.

Since JACQUES's type system cannot prove this for *any* field reference, *all* references would have to have guards in order to preserve safety. As a result, proving type soundness with guards is not markedly different from proving it with the incomplete context exception. Further, we conjecture that in practice, most programmers would specify null as their guard expression, thus hiding the incomplete context exception within a null pointer exception and thus severely hindering their own debugging efforts. Therefore, we have have chosen to omit guards and to raise an exception in cases where an object's environment does not suffice. If practical experience suggests that such guards are practically useful, we may add them to the language later.

# 6. SCALING TO FULL JAVA

To scale JACQUES's design to full Java [13] or C# [3], we must address two features of the larger language that our model does not include: concurrency and field shadowing. It seems that acquisition can co-exist with concurrency without any more than the normal difficulties associated with concurrent programs. In particular, if one thread changes the object containment graph while another is traversing that graph to look up an acquired field value, the results are undefined. It would be possible to define an acquired field reference to be an atomic operation via a Java synchronized block, but this seems unnecessary; the difficulties that can arise in situations like this are no worse than those which may arise in concurrent Java programs that involve heavy pointer manipulation.

Java-style field shadowing presents a more complicated challenge. In the case where an object acquires a shadowed or shadowing field from its container, it is no longer clear which field should be visible to the acquiring object. Consider the following class definitions:

```
class A extends Object ··· {X x ··· }
class B extends A ··· {X x ··· }
class C extends Object contained A {··· acquires X x ··· }
```

If c, an instance of C, is contained within b, an instance of B, it is most consistent with Java's current behavior if c acquires A.x rather than B.x, as c acquires properties from its container as viewed through A's interface. However, the language should provide the programmer with a means of overriding this default when necessary, much as Java allows the programmer to cast the object in a field reference expression to select which overloaded field is used.

# 7. RELATED WORK

The recent work in ownership types [2, 1] introduces constraints on the object composition graph in order to preserve certain desirable invariants, much as we do. In such a type system, the programmer annotates all field declarations, local variable declarations, method arguments, and method return types with ownership information. The type system then ensures that all paths from the root of the object containment graph to an object must pass through the object's owner. Consequently, if an object is owned by its container, only its container may refer to it. This restricts object aliasing and makes it easier to reason about object-oriented programs in the face of field mutation.

While ownership types and types for environmental acquisition both restrict the shape of the object containment graph, the similarities end there. While ownership constraints could help ensure that an object has at most one container, we believe that preventing aliasing to the degree achieved by their system is overly restrictive. In JACQUES, we explicitly allow an object to possess a reference to the contents of another object's contained field, so long as this additional reference does not imply a containment relationship. Further, we could remove the need for the "already contained" exception without complicating the type system further by disallowing direct assignment to contained fields and providing a switch-container operation that manipulates the object containment graph while preserving all desired invariants. Finally, ownership types do not appear to help our existing system's most severe drawback, namely the inability to detect at compile time whether an object will be in an environment that provides all of the necessary acquired fields and methods.

# 8. CONCLUSIONS AND FUTURE WORK

In this paper, we have resumed Gil and Lorenz's language design experiment in environmental acquisition. We have extended CLASSICJAVA to provide a formal operational semantics and a formal type system for a language that includes acquisition, thus placing the mechanism on a firm theoretical grounding.

We have also explored several of the design alternatives in the formal context, and we have found that type soundness means that many of these alternatives can only be resolved in certain ways. Soundness alone is not sufficient to resolve all of the design choices, however, so we clearly need more experience with acquisition to make the appropriate choices.

This research suggests three additional projects. First, we need to implement acquisition by modifying an existing class system, so that we can conduct program design experiments in the context of a full programming language. Second, we must search existing class libraries and frameworks for instances of environmental acquisition via pointer chasing or callbacks to collect a wide range of examples for this feature. Third, we must investigate the use of more advanced type systems for acquisition. These include the ability to

infer certain types, such as the list of possible containers for a class, and the use of resource-aware type systems to ensure that certain exceptions (most notably the "incomplete context" exception) cannot be generated.

*Acknowledgments:* We thank David Lorenz for taking the time to discuss his prior work with us. We also thank Mitchell Wand for his assistance with JACQUES's soundness proof. Finally, we thank the anonymous reviewers for their comments, insights, and suggestions.

## 9. REFERENCES

[1] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 292–310. ACM Press, 2002.

[2] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 48–64. ACM Press, 1998.

[3] Microsoft Corporation. *Microsoft C# Language Specifications*. Microsoft Press, 2001.

[4] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002. A preliminary version of this paper appeared in PLILP 1997, LNCS volume 1292, pp. 369–388.

[5] Matthew Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997. `http://www.plt-scheme.org/software/mzscheme/`.

[6] Matthew Flatt and Robert Bruce Findler. PLT MrEd: Graphical toolbox manual. Technical Report TR97-279, Rice University, 1997. `http://www.plt-scheme.org/software/mred/`.

[7] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, volume 1523 of *Springer Lecture Notes in Computer Science*, pages 241–269. Springer-Verlag, 1999. Preliminary version appeared in proceedings of *Principles of Programming Languages*, 1998. Revised version is Rice University technical report TR 97-293, June 1999.

[8] Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, MA, 1997.

[9] Jim Fulton. Extension classes, Python extension types become classes. `http://debian.acm.ndsu.nodak.edu/doc/python-extclass/ExtensionClass.html`.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[11] Joseph Gil and David H. Lorenz. Environmental acquisition: a new inheritance-like abstraction mechanism. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 214–231. ACM Press, 1996.

[12] Joseph (Yossi) Gil and David H. Lorenz. Object technology: Design patterns and language design. *IEEE Computer*, 31(3):118–120, March 1998.

[13] James Gosling, Bill Joy, and Guy Steele, Jr. *The Java Language Specification*. Addison-Wesley, 1996.

[14] Alan C. Kay. The early history of Smalltalk. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, pages 69–95. ACM Press, 1993.

[15] Shriram Krishnamurthi, Yan-David Erlich, and Matthias Felleisen. Expressing structural properties as language constructs. In *European Symposium on Programming*, volume 1576 of *Springer Lecture Notes in Computer Science*, pages 258–272, March 1999.

[16] Amos Latteier, Michel Pelletier, Chris McDonough, and Peter Sabaini. *The Zope Book*. SAMS, 2001. Also available on-line at `http://zope.org/Documentation/Books/ZopeBook`.

[17] Mark Logan, Matthias Felleisen, and David Blank-Edelman. Environmental acquisition in network management. In *Proceedings of LISA 2002: Sixteenth Systems Administration Conference*, pages 175–184. USENIX Association, 2002.

[18] Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *International Conference on Rewriting Techniques and Applications*, 2004.

[19] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[20] Sun Microsystems. Java 2 standard edition SDK, v1.4.2. `http://wwws.sun.com/software/communitysource/j2se/java2/download.html`.

[21] Sun Microsystems. Java foundation classes (JFC/Swing). `http://java.sun.com/products/jfc/index.jsp`.

[22] Python. `http://www.python.org/`.

[23] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. First appeared as Technical Report TR160, Rice University, 1991.