Environmental Acquisition Revisited

Richard Cobbe Matthias Felleisen

PLT at the College of Computer and Information Science
Northeastern University
Boston, MA 02115
cobbe@ccs.neu.edu

September 5, 2006

Northeastern University Technical Report NU-CCIS-06-10

Abstract

Object-oriented programs frequently deal with hierarchical forms of data, often called whole-part arrangements. When in such arrangements the methods in the parts must compute information about their container, the programmer is currently forced to manually maintain a two-way relationship, also known as friend pointers. The maintenance problem becomes particularly complex in the case of deeply nested hierarchies. To eliminate this drudgery of pointer maintainenace, Gil and Lorenz proposed a language construct, dubbed environmental acquisition, in 1996. The construct has been adapted and studied in several different contexts over the past decade.

In this paper, we analyze Gil and Lorenz's language design with a formal model. Specifically, we extend ClassicJava with environmental acquisition and determine what it takes to prove the revised type system sound. Then we study a number of design decisions in this framework and illustrate when a formal model can help (or not) make such decisions. We also discuss how this model helps us understand what a full-scale implementation of type-sound environmental acquisition would require.

1 From Patterns to Language Constructs

Object-oriented programs often manipulate data in the form of nested containers, which of course are equivalent to trees. For example, a program with a graphical user interface (GUI) tends to nest panels within panels inside a window to arrange the atomic widgets. Similarly, typical XML libraries create objects in the shape of trees to represent XML forms. In these contexts, it is common that methods compute information about the nesting of containers, and programming such a method is straightforward. If, however, the method in a containee of such a container hierarchy must compute data about its containers, programs become complicated. Specifically, the programs must establish and maintain knowledge about the container hierarchies at all times.

The problem of managing such hierarchies is the subject of several object-oriented design patterns [8]. In the particular case of nested hierarchies, a programmer should choose a composite pattern and maintain so-called "friend pointers." Those connect an item to its container and are therefore known as "two-way links" in the UML-oriented pattern community [6, p. 155]. Even though patterns help solve this problem, programmers must still maintain complex relationships among objects manually. For this reason, several people [10, 14] have proposed the idea that patterns suggest new constructs for programming languages. Then 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.

Gil and Lorenz [9] recognized the problem of *environmental acquisition*, i.e., the need for methods in containees to compute information about their context, and proposed constructs for specifying whole-part

```
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;
  }
  // ...
}
```

Figure 1: Root panes in Swing

relationships 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 [19, 7] as a part of the Zope project [15]. 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 [16].

In this paper—a revised and extended version of our POPL paper [2]—we resume and continue Gil and Lorenz's language design experiment. Specifically, we present a statically-typed version of environmental 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 [5], 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; we describe 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

Environmental acquisition is inseparably tied to the whole-part relationships of an object containment hierarchy. In many cases, the purpose of methods is to propagate information toward the root of such an object tree. Acquisition becomes relevant when an application must propagate information *away* from the root.

[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.

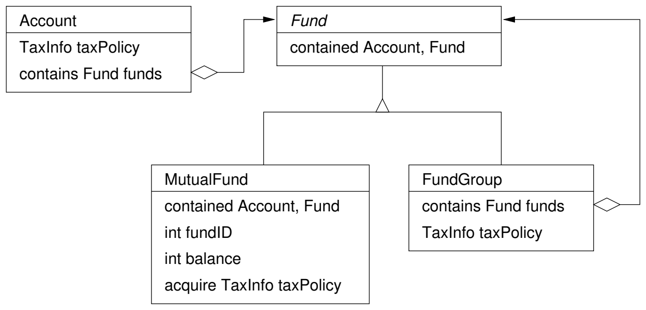


Figure 2: Acquisition example: investment accounts

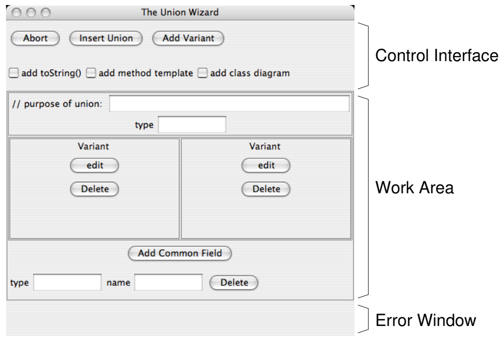


Figure 3: UnionInfo wizard dialog

Consider GUI frameworks, which are typically implemented with the composite pattern. In Java's Swing toolkit [18], for instance, all graphical components are subclasses of JComponent; certain 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. If Java supported environmental acquisition, each individual component could simply acquire a reference to the root pane from its container hierarchy, avoiding the need to write error-prone pointer-management code directly and instead leaving that responsibility to the language.

Following Gamma et al, we take our next example of acquisition from the financial sector. See figure 2 for the UML 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's ProfessorJ component [12], consider a class-writing wizard for use in a program development environment. This wizard allows a student to define a new class or set of classes by specifying their properties graphically; the produce method in the ClassUnionWizard class generates the corresponding code. ProfessorJ allows the student 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 figure 3 shows, each dialog contains three main visual components: a control interface at the top, an

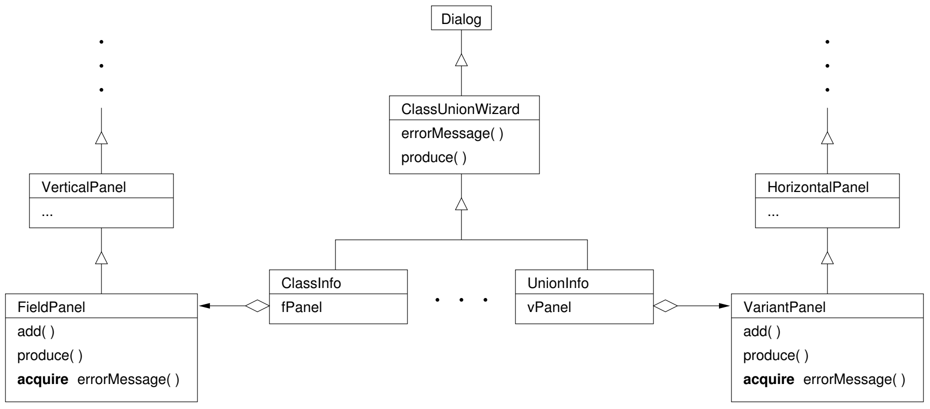


Figure 4: Acquisition example: class wizard

error window at the bottom, and a large work area in the center that is specialized to the specific wizard. 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. Figure 4 displays a UML diagram that summarizes the design.

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 cannot use inheritance to access the errorMessage method. Instead, 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 class 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 environmental 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. Unfortunately, both strategies involve significant bookkeeping. It is the purpose of environmental acquisition to eliminate such mundane tasks.

3 Classic Java

ClassicJava [5] is a formal model of the sequential core of Java version 1.4 [11] that includes inheritance, method overriding, and field mutation; the model could easily apply to similar OO languages, like C# [3] or Eiffel [17]. 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 semantics. In particular, 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 and assignments must know the static type of the object whose field is being accessed in order to resolve shadowed fields correctly.

Figure 6 describes the six type judgments necessary to define ClassicJava's static semantics; we refer

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

Figure 5: ClassicJava abstract syntax

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

Figure 6: ClassicJava type judgments

the reader to the original paper for the formal definition of these judgments. The ClassicJava semantics requires several additional relations; those that are relevant to our work are defined in figure 7. The dynamic semantics, a small-step operational semantics, defines reductions of the form $P \vdash \langle e, \mathcal{S}\rangle \hookrightarrow \langle e', \mathcal{S}'\rangle$, where P denotes the program's class definitions and $\mathcal{S}$ and $\mathcal{S}'$ are stores. 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., $c.\mathit{fd}$) to values.

ClassicJava satisfies a type soundness theorem:

Theorem 1 (1: Type Soundness for ClassicJava) *If* $\vdash_{\mathsf{p}} P \Rightarrow P' \ :\ t$ *and* $P' = \mathit{defn}_1\ \ldots\ \mathit{defn}_n\ e$, *then either*

- $P' \vdash \langle e, \varnothing\rangle \hookrightarrow^{*} \langle \mathit{obj}, \mathcal{S}\rangle$ *and* $\mathcal{S}(\mathit{obj}) = \langle t', \mathcal{F}\rangle$ *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 of ClassicJava that 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

$\prec_P$ immediate subclass

$c \prec_P c' \iff \textsf{class}\ c\ \textsf{extends}\ c'\ \{\cdots\}$ is in P

$\leqslant_P$ subclass

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

$\Subset_P$ field is declared in a class

$\langle c.fd, t\rangle \Subset_P c \iff \textsf{class}\ c\ \cdots\ \{\cdots\ t\ fd\ \cdots\}$ is in P

$\Subset_P$ method is declared in a class

$\langle md, (t_1 \ldots t_n \rightarrow t), (var_1 \ldots var_n), e\rangle \Subset_P c \iff$
$\quad \textsf{class}\ c\ \cdots\ \{\cdots\ t\ md(t_1\ var_1\ \ldots\ t_n\ var_n)\ \{e\}\ \cdots\}$ is in P

$\in_P$ field is contained in a class

$\langle c'.fd, t\rangle \in_P c \iff \langle c'.fd, t\rangle \Subset_P c'$ and $c' = \min\{c'' \mid c \leqslant_P c''$ and $\exists t' . \langle c''.fd, t'\rangle \Subset_P c''\}$

$\in_P$ method is contained in a class

$\langle md, T, V, e\rangle \in_P c \iff \langle md, T, V, e\rangle \Subset_P c'$ and
$\quad c' = \min\{c'' \mid c \leqslant_P c''$ and $\exists e', V' . \langle md, T, V', e'\rangle \in_P c''\}$

Figure 7: ClassicJava relations

and sketch a proof of type soundness for Jacques. In the following section, we discuss our design decisions and how the formal model helped making them.

Jacques subsumes almost all of ClassicJava, except for field shadowing. This simplifies the description of field acquisition. In section 6, we revisit the issue.

4.1 Syntax

To support acquisition along containment links between objects, we must have some way to distinguish between those fields 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.

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 cannot be contained (in the spirit of environmental acquisition) in any object.

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

Figure 8: JACQUES syntax extensions

$\textit{ClassFieldsOK}(P)$ no field definition shadows a field defined in a superclass

$(\langle t_1, \textit{fd}, s_1\rangle \Subset_P c_1 \text{ and } \langle t_2, \textit{fd}, s_2\rangle \Subset_P c_2 \text{ and } c_1 \neq c_2) \Longrightarrow c_1 \nleqslant_P c_2$

$\textit{containers}_P(c)$ set of all possible containers of instances of class c

$\textit{containers}_P(\textsf{Object}) = \varnothing$

$\textit{containers}_P(c) = \{c_1, \ldots, c_n\}$ where $\textsf{class } c \cdots \textsf{ contained } c_1 \ldots c_n \ \{ \cdots \}$ is in P

$\textit{acqFields}_P(c)$ set of all acquired fields in class c

$\textit{acqFields}_P(\textsf{Object}) = \varnothing$

$\textit{acqFields}_P(c) = \{\langle t, \textit{fd}, \textsf{acquired}\rangle \mid \langle t, \textit{fd}, \textsf{acquired}\rangle \in_P c\}$
where $\textsf{class } c \cdots \{ \cdots \}$ is in P

$\textit{acqMethods}_P(c)$ set of all acquired methods in class c

$\textit{acqMethods}_P(\textsf{Object}) = \varnothing$

$\textit{acqMethods}_P(c) = \{\langle \textit{md}, T, V, e, \textsf{acquired}\rangle \mid \langle \textit{md}, T, V, e, \textsf{acquired}\rangle \in_P c\}$
where $\textsf{class } c \cdots \{ \cdots \}$ is in P

$\sqsubset_P$ class may be contained

$c_1 \sqsubset_P c_2 \Longleftrightarrow$ there exists $c' \in \textit{containers}_P(c_1)$ such that $c_2 \leqslant_P c'$

Figure 9: JACQUES relations

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 $\langle t, \textit{fd}, s\rangle$, where t is the field's type, $\textit{fd}$ 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 $\langle \textit{md}, T, V, e, s\rangle$, where $\textit{md}$, T, V, and e are the method's name, type signature, argument names, and body expression, as in CLASSICJAVA, 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 $\in_P$ and other relations.)

Finally, we also use an adjusted instance representation: $\langle c, \textit{ctnr}, \mathcal{F}\rangle$ where c is the class tag, $\textit{ctnr}$ 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 $\Subset_P$ and $\in_P$. The additional necessary predicates are straightforward; their definitions are provided in figures 9 and 10.

We use the idea of the class containment graph G_P to describe the possible object containment relationships in a JACQUES program. Figure 10 provides the formal definition of G_P. Intuitively, there is a path from class c_1 to class c_2 in G_P if an instance of c_1 can be contained in an instance of c_2 according to the static annotations in the program source. (Not all paths in G_P correspond to potential containment relationships, but these extra paths do not present any problems.)

The predicates $\textit{canAcq}_P$ and $\textit{canAcqMeth}_P$, also defined in figure 10, specify whether a class c may safely acquire a specific field or method. The class c may safely acquire the field $\textit{fd}$ with type t if one of the

$<:_P$ subtyping

$t <:_P t' \iff t \leqslant_P t'$

$(t_1 \ldots t_m \to t) <:_P (t'_1 \ldots t'_n \to t') \iff (m = n \text{ and } t <:_P t' \text{ and } t'_i <:_P t_i \text{ for all } i \in [1, n])$

G_P Class containment graph

$G_P = (V_P, E_P)$

where $V_P = \mathrm{dom}(\prec_P) \cup \{\mathsf{Object}\}$

and $E_P = \{(c_1, c_2) \in (V_P)^2 \mid c_2 \in \mathit{containers}_P(c_1) \text{ or } c_2 \prec_P c_1\}$

$\mathit{canAcq}_P(c, \langle t, \mathit{fd}, s\rangle)$ class c may safely acquire field fd with type t from its containers

```
for all paths $c_0, c_1, \ldots$ in $G_P$, where $c_0 = c$
  if there exists an $i \geqslant 1$ such that $\langle t', \mathit{fd}, \_\rangle \in_P c_i$ and $t' \not<:_P t$
    then there must exist a $j \in [1, i)$ such that
      $\langle t'', \mathit{fd}, s\rangle \in_P c_j$ and $t'' <:_P t$ and $s \neq \mathsf{acquired}$.
```

$\mathit{canAcqMeth}_P(c, \langle \mathit{md}, (t_1 \ldots t_n \to t), V, e, s\rangle)$

(class c can safely acquire method md with the given type)

```
for all paths $c_0, c_1, \ldots$ in $G_P$, where $c_0 = c$
  if there exists an $i \geqslant 1$ such that $\langle \mathit{md}, (t'_1 \ldots t'_m \to t'), \_, \_, \_\rangle \in_P c_i$ and
                                              $t'_1 \ldots t'_m \to t' \not<:_P t_1 \ldots t_n \to t$
    then there must exist a $j \in [1, i)$ such that
      $\langle \mathit{md}, (t''_1 \ldots t''_n \to t''), \_, \_, s\rangle \in_P c_j$ and
      $(t''_1 \ldots t''_n \to t'') <:_P (t_1 \ldots t_n \to t)$ and
      $s \neq \mathsf{acquired}$.
```

Figure 10: Acquisition safety conditions

Prog

$$\frac{\begin{array}{c} \mathit{ClassesOnce}(P) \qquad \mathit{MethodOncePerClass}(P) \qquad \mathit{FieldOncePerClass}(P) \\ \mathit{CompleteClasses}(P) \qquad \mathit{WellFoundedClasses}(P) \qquad \mathit{ClassMethodsOK}(P) \qquad \mathit{ClassFieldsOK}(P) \\ P \vdash_{\mathsf{d}} \mathit{defn}_i \Rightarrow \mathit{defn}'_i \text{ for all } i \in [1,n] \qquad P, \varnothing \vdash_{\mathsf{e}} e \Rightarrow e' \ : \ t \qquad \text{where } P = \mathit{defn}_1 \ldots \mathit{defn}_n \ e \end{array}}{\vdash_{\mathsf{p}} \mathit{defn}_1 \ldots \mathit{defn}_n \ e \Rightarrow \mathit{defn}'_1 \ldots \mathit{defn}'_n \ e' \ : \ t}$$

Defn

$$\frac{\begin{array}{c} P \vdash_{\mathsf{t}} t_j \text{ for all } j \in [1,m] \qquad P \vdash_{\mathsf{t}} c_j \text{ for all } j \in [1,n] \qquad P, c \vdash_{\mathsf{m}} \mathit{meth}_i \Rightarrow \mathit{meth}'_i \text{ for all } i \in [1,p] \\ \forall f \in \mathit{acqFields}_P(c) \,.\, \mathit{canAcq}_P(c,f) \qquad \forall m \in \mathit{acqMethods}_P(c) \,.\, \mathit{canAcqMeth}_P(c,m) \\ \mathit{containers}_P(c') \subseteq \mathit{containers}_P(c) \qquad t_i \sqsubset_P c \text{ for all } i \in [k+1, \ell] \end{array}}{\begin{array}{lll} P \vdash_{\mathsf{d}} \ \mathsf{class}\ c\ \mathsf{extends}\ c'\ \mathsf{contained}\ c_1, \ldots, c_n & \Rightarrow & \mathsf{class}\ c\ \mathsf{extends}\ c'\ \mathsf{contained}\ c_1, \ldots, c_n \\ \quad \{t_1\ \mathit{fd}_1 \ldots t_k\ \mathit{fd}_k & & \{t_1\ \mathit{fd}_1 \ldots t_k\ \mathit{fd}_k \\ \quad\ \mathsf{contains}\ t_{k+1}\ \mathit{fd}_{k+1} \ldots t_\ell\ \mathit{fd}_\ell & & \ \mathsf{contains}\ t_{k+1}\ \mathit{fd}_{k+1} \ldots t_\ell\ \mathit{fd}_\ell \\ \quad\ \mathsf{acquires}\ t_{\ell+1}\ \mathit{fd}_{\ell+1} \ldots t_m\ \mathit{fd}_m & & \ \mathsf{acquires}\ t_{\ell+1}\ \mathit{fd}_{\ell+1} \ldots t_m\ \mathit{fd}_m \\ \quad\ \mathit{meth}_1 \ldots \mathit{meth}_p & & \ \mathit{meth}'_1 \ldots \mathit{meth}'_p \\ \quad\ \mathsf{acquires}\ \mathit{acqmeth}_1 \ldots \mathit{acqmeth}_q\} & & \ \mathsf{acquires}\ \mathit{acqmeth}_1 \ldots \mathit{acqmeth}_q\} \end{array}}$$

Method

$$\frac{P \vdash_{\mathsf{t}} t \qquad P \vdash_{\mathsf{t}} t_i \text{ for all } i \in [1,n] \qquad P, \{\mathsf{this} \mapsto t_0, \mathit{var}_1 \mapsto t_1, \ldots, \mathit{var}_n \mapsto t_n\} \vdash_{\mathsf{s}} e \Rightarrow e' \ : \ t}{P, t_0 \vdash_{\mathsf{m}} t\ \mathit{md}(t_1\ \mathit{var}_1 \ldots t_n\ \mathit{var}_n)\ \{e\} \Rightarrow t\ \mathit{md}(t_1\ \mathit{var}_1 \ldots t_n\ \mathit{var}_n)\ \{e'\}}$$

Subtype

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' \ : \ t \qquad t' <:_P t}{P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' \ : \ t'}$$

Type Existence

$$\frac{t \in \mathrm{dom}(\prec_P) \cup \{\mathsf{Object}\}}{P \vdash_{\mathsf{t}} t}$$

Figure 11: Jacques type elaboration rules

following conditions holds for all paths in the class containment graph that begin with c:

1. The field *fd* is not defined in any class on the path. If any of their subclasses define *fd*, they define it with a compatible type.
2. Alternatively, the first class on the path that defines the field *fd* defines it with a compatible type. Further, if *fd* is defined in any subclass of a class that appears earlier on the path, it must be defined with a compatible type.

The class c may acquire the method *md* under similar circumstances; the only significant difference is in the notion of a "compatible type."

Surprisingly, these relations allow the programmer to define a class that acquires a field that is not provided by any class in the program. While this may seem overly permissive, it is the most reasonable solution. On the one hand, it has proven challenging to formulate alternative canAcq_P and $\mathit{canAcqMeth}_P$ predicates that accomodate cycles in the class containment graph but forbid paths without matches. On the other hand, even if Jacques performed these stronger static checks, objects could still have null parent pointers, and the run-time system would still have to perform appropriate checks.

Figures 11 and 12 provide Jacques's type rules. These are based on ClassicJava's type rules, with modifications to allow acquisition. Because Jacques does not support field shadowing, neither the [Get] nor the [Set] rules annotate field reference or assignment expressions, and we add $\mathit{ClassFieldsOK}(P)$ to [Prog]'s antecedents.[2] Further, [Set] prohibits assignments to acquired fields; see section 5.5 for the rationale for this restriction.

The [Defn] rule contains most of the changes that support acquisition. First, it verifies that class c's containers provide the necessary fields and methods with compatible types. Second, c's containers must be

[2]The other predicates in the [Prog] rule are taken *mutatis mutandis* from ClassicJava.

$$\textsc{Ctor}\quad \frac{P \vdash_{\mathsf{t}} c}{P, \Gamma \vdash_{\mathsf{e}} \mathsf{new}\ c \Rrightarrow \mathsf{new}\ c\ :\ c}$$

$$\textsc{VarRef}\quad \frac{var \in \mathrm{dom}(\Gamma) \qquad \Gamma(var) = t}{P, \Gamma \vdash_{\mathsf{e}} var \Rrightarrow var\ :\ t}$$

$$\textsc{Null}\quad \frac{P \vdash_{\mathsf{t}} t}{P, \Gamma \vdash_{\mathsf{e}} \mathsf{null} \Rrightarrow \mathsf{null}\ :\ t}$$

$$\textsc{FieldRef}\quad \frac{P, \Gamma \vdash_{\mathsf{e}} e \Rrightarrow e'\ :\ t' \qquad \langle t, fd, \_ \rangle \in_P t'}{P, \Gamma \vdash_{\mathsf{e}} e.fd \Rrightarrow e'.fd\ :\ t}$$

$$\textsc{FieldSet}\quad \frac{\begin{array}{c} P, \Gamma \vdash_{\mathsf{s}} e_1 \Rrightarrow e'_1\ :\ t' \qquad \langle t, fd, s \rangle \in_P t' \\ s \neq \mathsf{acquired} \qquad P, \Gamma \vdash_{\mathsf{s}} e_2 \Rrightarrow e'_2\ :\ t \end{array}}{P, \Gamma \vdash_{\mathsf{e}} e_1.fd \mathrel{=} e_2 \Rrightarrow e'_1.fd \mathrel{=} e'_2\ :\ t}$$

$$\textsc{Call}\quad \frac{P, \Gamma \vdash_{\mathsf{e}} e \Rrightarrow e'\ :\ c \qquad P, \Gamma \vdash_{\mathsf{s}} e_i \Rrightarrow e'_i\ :\ t_i \text{ for all } i \in [1, n] \qquad \langle md, (t_1 \dots t_n \rightarrow t), \_, \_, \_ \rangle \in_P c}{P, \Gamma \vdash_{\mathsf{e}} e.md(e_1 \dots e_n) \Rrightarrow e'.md(e'_1 \dots e'_n)\ :\ t}$$

$$\textsc{Super}\quad \frac{\begin{array}{c} P, \Gamma \vdash_{\mathsf{e}} \mathsf{this} \Rrightarrow \mathsf{this}\ :\ c' \qquad c' \prec_P c \\ \langle md, (t_1 \dots t_n \rightarrow t), \_, \_, \_ \rangle \in_P c \qquad P, \Gamma \vdash_{\mathsf{s}} e_i \Rrightarrow e'_i\ :\ t_i \text{ for all } i \in [1, n] \end{array}}{P, \Gamma \vdash_{\mathsf{e}} \mathsf{super}\,.\, md(e_1 \dots e_n) \Rrightarrow \mathsf{super}\underline{\equiv \mathsf{this} : c}.md(e'_1 \dots e'_n)\ :\ t}$$

$$\textsc{WideCast}\quad \frac{P, \Gamma \vdash_{\mathsf{s}} e \Rrightarrow e'\ :\ t}{P, \Gamma \vdash_{\mathsf{e}} \mathsf{cast}\ t\ e \Rrightarrow \mathsf{cast}\ t\ e'\ :\ t}$$

$$\textsc{NarrowCast}\quad \frac{P, \Gamma \vdash_{\mathsf{e}} e \Rrightarrow e'\ :\ t' \qquad t' \leqslant_P t}{P, \Gamma \vdash_{\mathsf{e}} \mathsf{cast}\ t\ e \Rrightarrow \mathsf{cast}\ t\ e'\ :\ t}$$

$$\textsc{Let}\quad \frac{P, \Gamma \vdash_{\mathsf{e}} e_1 \Rrightarrow e'_1\ :\ t_1 \qquad P, \Gamma[var \mapsto t_1] \vdash_{\mathsf{e}} e_2 \Rrightarrow e'_2\ :\ t}{P, \Gamma \vdash_{\mathsf{e}} \mathsf{let}\ var \mathrel{=} e_1\ \mathsf{in}\ e_2 \Rrightarrow \mathsf{let}\ var \mathrel{=} e'_1\ \mathsf{in}\ e'_2\ :\ t}$$

Figure 12: Jacques type elaboration rules, continued

a superset of c's superclass's containers, to preserve behavioral subtyping.[3] Third, this rule ensures 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 [Defn] rule could also verify that class A can contain instances of class B if and only if B can be contained in A, and that class A may be contained in at least one other class if it acquires any fields or methods. While warnings about such violations are likely to be useful to the programmer, they are not required for type soundness.

4.3 Semantics

Jacques's semantics is based on that of ClassicJava. Figure 13 defines the Jacques reduction relation; figure 14 provides additional supplementary functions. These reductions also perform run-time checks to ensure that acquisition is well-defined. First, we require that there be no containment cycles in the program; the [*ct-set*] and [*cycle-set*] rules enforce this invariant and signal a "container cycle" if it is violated. Second, the [*ct-set*] and [*x-ct-set*] rules ensure that, on assignment to a contained field, the field's new contents are not already contained elsewhere; they signal an "already contained" error otherwise. Finally, because any object's container reference can be null, the [*aget*], [*acall*], and [*asuper*] rules must ensure the presence of a container that provides the acquired field or method; the [*xaget*], [*xacall*], and [*xasuper*] rules signal an "incomplete context" error otherwise.

4.4 Jacques Soundness

Jacques also satisfies a type soundness theorem:

Theorem 2 (2: Type Soundness for Jacques) *If*

$$\vdash_{\mathsf{p}} P \Rightarrow P' \ : \ t \text{ and } P' = defn_1 \ \dots \ defn_n \ e,$$

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 arise from 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 arrays; many errors can be caught statically, but some run-time checks remain necessary.

The soundness proof follows the standard method [20] as applied to ClassicJava [5]. In addition to the subject reduction and progress lemmas, we also require some definitions and supporting lemmas beyond those used in the ClassicJava soundness proof; they are stated below.

[3]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 is simply a trade-off that the programmer must consider during the design of class hierarchies.

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

$P \vdash \langle \mathsf{E}[\mathsf{new}\ c], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[obj], \mathcal{S}[obj \mapsto \langle c, \mathsf{null}, \mathcal{F}\rangle]\rangle$ [*new*]
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*]
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*]
where $\langle t, fd, \mathsf{acquired}\rangle \in_P \mathit{classTag}(\mathcal{S}(obj))$
and $\mathit{getAcqField}_P(\mathcal{S}, obj, fd) \neq \mathsf{notFound}$

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

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

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

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

$P \vdash \langle \mathsf{E}[obj.fd\ \mathsf{=}\ v], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error\colon\ container\ cycle}, \mathcal{S}\rangle$ [*cycle-set*]
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 $\mathit{canReach}_P(\mathcal{S}, obj, v)$

$P \vdash \langle \mathsf{E}[obj.fd\ \mathsf{=}\ v], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathit{updateCtdField}(\mathcal{S}, obj, fd, v)\rangle$ [*ct-set*]
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 $\neg \mathit{canReach}_P(\mathcal{S}, obj, v)$

$P \vdash \langle \mathsf{E}[obj.md(v_1\ \ldots\ v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e_b[obj/\mathsf{this}, v_1/var_1, \ldots, v_n/var_n]], \mathcal{S}\rangle$ [*call*]
where $\langle md, \_, (var_1, \ldots, var_n), e_b, \mathsf{normal}\rangle \in_P \mathit{classTag}(\mathcal{S}(obj))$

$P \vdash \langle \mathsf{E}[obj.md(v_1\ \ldots\ v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e_b[obj'/\mathsf{this}, v_1/var_1, \ldots, v_n/var_n]], \mathcal{S}\rangle$ [*acall*]
where $\langle md, \_, \_, e, \mathsf{acquired}\rangle \in_P \mathit{classTag}(\mathcal{S}(obj))$
and $\langle obj', (var_1, \ldots, var_n), e_b\rangle = \mathit{getAcqMethod}_P(\mathcal{S}, obj, md)$

$P \vdash \langle \mathsf{E}[obj.md(v_1\ \ldots\ v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error\colon\ incomplete\ context}, \mathcal{S}\rangle$ [*xacall*]
where $\langle md, \_, \_, e, \mathsf{acquired}\rangle \in_P \mathit{classTag}(\mathcal{S}(obj))$
and $\mathit{getAcqMethod}_P(\mathcal{S}, obj, md) = \mathsf{notFound}$

$P \vdash \langle \mathsf{E}[\mathsf{super}\underline{\equiv obj : c'}.md(v_1\ \ldots\ v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[obj/\mathsf{this}, v_1/var_1, \ldots, v_n/var_n]], \mathcal{S}\rangle$ [*super*]
where $\langle md, \_, (var_1, \ldots, var_n), e, \mathsf{normal}\rangle \in_P c'$

$P \vdash \langle \mathsf{E}[\mathsf{super}\underline{\equiv obj : c'}.md(v_1\ \ldots\ v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[obj'/\mathsf{this}, v_1/var_1, \ldots, v_n/var_n]], \mathcal{S}\rangle$ [*asuper*]
where $\langle md, \_, \_, \_, \mathsf{acquired}\rangle \in_P c'$
and $\langle obj', (var_1, \ldots, var_n), e\rangle = \mathit{getAcqMethod}_P(\mathcal{S}, obj, md)$

$P \vdash \langle \mathsf{E}[\mathsf{super}\underline{\equiv obj : c'}.md(v_1\ \ldots\ v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error\colon\ incomplete\ context}, \mathcal{S}\rangle$ [*xasuper*]
where $\langle md, \_, \_, \_, \mathsf{acquired}\rangle \in_P c'$ and $\mathit{getAcqMethod}_P(\mathcal{S}, obj, md) = \mathsf{notFound}$

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

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

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

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

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

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

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

Figure 13: Jacques operational semantics

$$
\begin{array}{rcl}
\mathit{getAcqField}_P(\mathcal{S}, \mathsf{null}, \mathit{fd}) & = & \mathsf{notFound} \\
\mathit{getAcqField}_P(\mathcal{S}, \mathit{obj}, \mathit{fd}) & = & \begin{cases} \mathcal{F}(\mathit{fd}) & \text{if } \mathit{fd} \in \mathrm{dom}(\mathcal{F}) \\ \mathit{getAcqField}_P(\mathcal{S}, \mathit{ctnr}, \mathit{fd}) & \text{otherwise} \end{cases} \\
 & & \quad \text{where } \langle \_, \mathit{ctnr}, \mathcal{F} \rangle = \mathcal{S}(\mathit{obj}) \\
 & & \\
\mathit{getAcqMethod}_P(\mathcal{S}, \mathsf{null}, \mathit{md}) & = & \mathsf{notFound} \\
\mathit{getAcqMethod}_P(\mathcal{S}, \mathit{obj}, \mathit{md}) & = & \begin{cases} \langle \mathit{obj}, (\mathit{var}_1, \ldots, \mathit{var}_n), e_b \rangle \\ \qquad \text{if } \langle \mathit{md}, \_, (\mathit{var}_1, \ldots, \mathit{var}_n), e_b, \mathsf{normal} \rangle \in_P c \\ \mathit{getAcqMethod}_P(\mathcal{S}, \mathit{ctnr}, \mathit{md}) \quad \text{otherwise} \end{cases} \\
 & & \quad \text{where } \langle c, \mathit{ctnr}, \_ \rangle = \mathcal{S}(\mathit{obj}) \\
 & & \\
\mathit{updateField}(\mathcal{S}, \mathit{obj}, \mathit{fd}, v) & = & \mathcal{S}[\mathit{obj} \mapsto \langle c, \mathit{ctnr}, \mathcal{F}[\mathit{fd} \mapsto v] \rangle] \\
 & & \quad \text{where } \langle c, \mathit{ctnr}, \mathcal{F} \rangle = \mathcal{S}(\mathit{obj}) \\
 & & \\
\mathit{updateCtdField}(\mathcal{S}, \mathit{obj}, \mathit{fd}, v) & = & \mathit{updateField}(\mathcal{S}_2, \mathit{obj}, \mathit{fd}, v) \\
 & & \quad \text{where } \langle c, \mathit{ctnr}, \mathcal{F} \rangle = \mathcal{S}(\mathit{obj}) \text{ and} \\
 & & \qquad \mathcal{S}_1 = \mathit{updateContainer}(\mathcal{S}, \mathcal{F}(\mathit{fd}), \mathsf{null}) \text{ and} \\
 & & \qquad \mathcal{S}_2 = \mathit{updateContainer}(\mathcal{S}_1, v, \mathit{obj}) \\
 & & \\
\mathit{updateContainer}(\mathcal{S}, v, \mathit{ctnr}) & = & \begin{cases} \mathcal{S} & \text{if } v = \mathsf{null} \\ \mathcal{S}[v \mapsto \langle c, \mathit{ctnr}, \mathcal{F} \rangle] & \text{otherwise} \\ \quad \text{where } v \in \mathrm{dom}(\mathcal{S}) \text{ and } \langle c, \_, \mathcal{F} \rangle = \mathcal{S}(v) \end{cases} \\
 & & \\
\mathit{canReach}_P(\mathcal{S}, \mathit{obj}, v) & \iff & \mathit{obj} = v \text{ or } (\mathit{ctnr} \neq \mathsf{null} \text{ and } \mathit{canReach}_P(\mathcal{S}, \mathit{ctnr}, v)) \\
 & & \quad \text{where } \mathit{ctnr} = \mathit{ctnr}(\mathcal{S}(\mathit{obj}))
\end{array}
$$

Figure 14: Jacques supplementary functions

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:*

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

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

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

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 $P, \Gamma \vdash_\sigma \mathcal{S}$ if the type environment Γ and the store $\mathcal{S}$ are consistent with one another, given program P. Formally:*

$$\begin{array}{lll} P, \Gamma \vdash_\sigma \mathcal{S} \iff & & \\ \quad obj \in \mathrm{dom}(\Gamma) \implies obj \in \mathrm{dom}(\mathcal{S}) & & (esc\text{-}1) \\ \wedge \quad \mathrm{dom}(\mathcal{S}) \subseteq \mathrm{dom}(\Gamma) & & (esc\text{-}2) \\ \wedge \quad G_{\mathcal{S}} \text{ has no cycles and is finite} & & (esc\text{-}3) \\ \wedge \quad [\mathcal{S}(obj) = \langle c, ctnr, \mathcal{F} \rangle \implies & & \\ \qquad \Gamma(obj) = c & & (esc\text{-}4) \\ \quad \wedge \quad \mathrm{dom}(\mathcal{F}) = \{fd \mid \langle \_, fd, s \rangle \in_P c, s \in \{\mathsf{normal}, \mathsf{contained}\}\} & & (esc\text{-}5) \\ \quad \wedge \quad \mathrm{rng}(\mathcal{F}) \subseteq \mathrm{dom}(\mathcal{S}) \cup \{\mathsf{null}\} & & (esc\text{-}6) \\ \quad \wedge \quad (\mathcal{F}(fd) \in \mathrm{dom}(\mathcal{S}) \text{ and } \langle c', fd, \_ \rangle \in_P c) \implies & & (esc\text{-}7) \\ \qquad\quad classTag(\mathcal{S}(\mathcal{F}(fd))) \leqslant_P c' & & \\ \quad \wedge \quad (\mathcal{F}(fd) = \mathcal{F}(fd') \neq \mathsf{null} \text{ and } \langle t, fd, \mathsf{contained} \rangle \in_P c \text{ and} & & (esc\text{-}8) \\ \qquad\quad \langle t', fd', \mathsf{contained} \rangle \in_P c) \implies fd = fd' & & \\ \quad \wedge \quad ctnr \in \mathrm{dom}(\mathcal{S}) \iff P, \mathcal{S} \vdash ctnr \sqsupseteq obj & & (esc\text{-}9) \\ \quad \wedge \quad ctnr \in \mathrm{dom}(\mathcal{S}) \implies & & (esc\text{-}10) \\ \qquad\quad \exists c' \in containers_P(c) \,.\, classTag(\mathcal{S}(ctnr)) \leqslant_P c' & & \\ \quad \wedge \quad ctnr \in \mathrm{dom}(\mathcal{S}) \cup \{\mathsf{null}\}] & & (esc\text{-}11) \end{array}$$

Most of the clauses of this definition are present in ClassicJava; clauses 3 and 8–11 are new for Jacques. In order, they place the following requirements on the store:

- (*esc-3*): The object containment graph must be a finite set of finite trees. (The definition of $G_{\mathcal{S}}$ also implies that these trees are disjoint.)
- (*esc-8*): No object may have two contained fields that refer to the same object. This allows us to set an object's container reference to null when we remove it from its container without breaking any other invariants.
- (*esc-9*): The object *obj*'s container reference is *ctnr* if and only if *ctnr* contains *obj*. (As a consequence, if *ctnr* is null, then no object may contain *obj*.)
- (*esc-10*): 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.
- (*esc-11*): An object's container reference is either a valid object reference or null.

Lemma 1 *Let P be a well-typed program, and let Γ and $\mathcal{S}$ be a type environment and store, respectively, such that $P, \Gamma \vdash_\sigma \mathcal{S}$. For every path from obj_1 to obj_n in $G_\mathcal{S}$, where $n > 1$, there exists a path $c_1, \ldots, c_2, \ldots, c_n$ in G_P, such that $\mathcal{S}(obj_i) = \langle c_i, \_, \_ \rangle$ for all $i \in [1, n]$, and such that for all $i \in [1, n)$, we can decompose the sub-path $c_i, \ldots, c_{i+1}$ into $c_{i0}, c_{i1}, \ldots, c_{im}$, where $c_{i0} = c_i$ and $c_{im} = c_{i+1}$ and $c_{i1} \in containers_P(c_{i0})$ and $c_{i(j+1)} \prec_P c_{ij}$ for all $j \in [2, m)$. We say that any path in G_P that corresponds to a path in $G_\mathcal{S}$ and that decomposes in this manner is* well-structured.

Proof 1 Strong induction on n, the length of the path.

Base case: $n = 2$.

Let $\langle c_1, \_, \_ \rangle = \mathcal{S}(obj_1)$ and $\langle c_2, \_, \_ \rangle = \mathcal{S}(obj_2)$. By (*esc-10*), there exists a class $c' \in containers_P(c_1)$ such that $c_2 \leqslant_P c'$. Since $\leqslant_P$ is the reflexive-transitive closure of $\prec_P$, there exists a sequence of classes $c'_1, \ldots, c'_m$ such that $c_2 \prec_P c'_m \prec_P \cdots \prec_P c'_1$ and $c'_1 = c'$. By construction of G_P, then, $c_1, c'_1, \ldots, c'_m, c_2$ is a path in G_P, and by the definition above, it is well-structured.

Inductive step: Decompose the path $obj_1, \ldots, obj_n$ into two paths obj_1, obj_2 and $obj_2, \ldots, obj_n$. By induction, there exist well-structured paths $c_1, \ldots, c_2$ and $c_2, \ldots, c_n$ in G_P, such that $\mathcal{S}(obj_1) = \langle c_1, \_, \_ \rangle$, $\mathcal{S}(obj_2) = \langle c_2, \_, \_ \rangle$, and $\mathcal{S}(obj_n) = \langle c_n, \_, \_ \rangle$. The concatenation of these paths $c_1, \ldots, c_2, \ldots, c_n$ is a path in G_P, and it is also well-structured. ■

Lemma 2 *Let P be a well-typed program, and let Γ and $\mathcal{S}$ be a type environment and a store, respectively, such that $P, \Gamma \vdash_\sigma \mathcal{S}$. If*

- $\mathcal{S}(obj) = \langle c, \_, \_ \rangle$, *and*
- $\langle t, fd, \mathsf{acquired} \rangle \in_P c$, *and*
- $canAcq_P(c, \langle t, fd, \mathsf{acquired} \rangle)$, *and*
- $getAcqField_P(\mathcal{S}, obj, fd) \Downarrow v$ *where* $v \neq \mathsf{notFound}$,

then either

- $v = \mathsf{null}$, *or*
- $v \in \mathrm{dom}(\mathcal{S})$ *and* $\mathcal{S}(v) = \langle c_v, \_, \_ \rangle$ *and* $c_v <:_P t$.

Similarly, if

- $\mathcal{S}(obj) = \langle c, \_, \_ \rangle$, *and*
- $\langle md, (t_1 \ldots t_n \to t), V, e, \mathsf{acquired} \rangle \in_P c$, *and*
- $canAcqMeth_P(c, \langle md, (t_1 \ldots t_n \to t), V, e, \mathsf{acquired} \rangle)$, *and*
- $getAcqMethod_P(\mathcal{S}, obj, md) \Downarrow \langle v, (var_1, \ldots, var_n), e_b \rangle$,

then

- $\mathcal{S}(v) = \langle c_v, \_, \_ \rangle$, *and*
- $\langle md, (t'_1 \ldots t'_n \to t'), (var_1, \ldots, var_n), e_b, \_ \rangle \in_P c_v$, *and*
- $(t'_1 \ldots t'_n \to t') <:_P (t_1 \ldots t_n \to t)$.

Proof 2 We provide the proof only for fields; that for methods is similar.

Since $getAcqField_P(\mathcal{S}, obj, fd) \Downarrow v$ and $v \neq \mathsf{notFound}$, there must exist a path $obj_0, obj_1, \ldots, obj_n$ in $G_\mathcal{S}$, where $obj_0 = obj$ and $\mathcal{S}(obj_i) = \langle c_i, \_, \_ \rangle$ for all $i \in [0, n]$ and $\langle \_, fd, s_i \rangle \in_P c_i \implies s_i = \mathsf{acquired}$ for all $i \in [0, n)$ and $\langle t', fd, s' \rangle \in_P c_n$ and $s' \neq \mathsf{acquired}$.

By lemma 1, there exists a well-structured path $c_0, \ldots, c_1, \ldots, c_n$ in G_P, where $c_0 = c$. Assume for the sake of a later contradiction that $t' \not<:_P t$. Then $canAcq_P(c, \langle t, fd, \mathsf{acquired} \rangle)$ requires that there be some class $c' \neq c_n$ on this well-structured path such that $\langle t'', fd, s \rangle \in_P c'$, $t'' <:_P t$, and $s \neq \mathsf{acquired}$. Because $\langle t, fd, \mathsf{acquired} \rangle \in_P c_0$, we know that $c' \neq c_0$. Therefore, three cases remain:

1. $c' = c_i$ for some $i \in [1, n)$. This contradicts our earlier conclusion that none of the intervening classes $c_1, \ldots, c_{n-1}$ define the field fd.

$$
\begin{array}{rcl}
\mathit{redex} & = & \mathsf{new}\ c \mid v.\mathit{fd} \mid v.\mathit{fd} = v \mid v.\mathit{md}(v^*) \\
& \mid & \mathsf{super}\underline{\ \equiv v : c}.\mathit{md}(v^*) \mid \mathsf{cast}\ t\ v \mid \mathsf{let}\ \mathit{var} = v\ \mathsf{in}\ e
\end{array}
$$

Figure 15: Jacques redexes

2. $c' = c_n$. This contradicts our assumption that c_n defines the field with an incompatible type.

3. c' is on a path $c_i, \ldots, c_{i+1}$ for some $i \in [0, n)$, but $c' \neq c_i$ and $c' \neq c_{i+1}$. Because this path is well-structured, $c_{i+1} \leqslant_P c'$. Because P is well-typed, then $\mathit{ClassFieldsOK}(P)$ must hold, which requires that $\langle t'', \mathit{fd}, s\rangle \in_P c_{i+1}$. This violates our earlier assumptions. If $i + 1 < n$, then an intermediate class provides the field. Otherwise, $i + 1 = n$, which violates our assumption that c_n provides the field with an incompatible type.

Therefore, our assumption $t' \not<:_P t$ must be false, and $t' <:_P t$. Then $v = \mathcal{F}_n(\mathit{fd})$, where $\mathcal{S}(\mathit{obj}_n) = \langle c_n, \_, \mathcal{F}_n\rangle$. If $\mathcal{F}_n(\mathit{fd}) = \mathsf{null}$, the lemma is satisfied. Otherwise, $\mathcal{F}_n(\mathit{fd}) \in \mathrm{dom}(\mathcal{S})$ by (*esc-6*), and (*esc-7*) then requires that $\mathcal{S}(\mathcal{F}_n(\mathit{fd})) = \langle c_v, \_, \_\rangle$, where $c_v <:_P t'$. This implies $c_v <:_P t$, as required. ■

The next lemma uses the predicate *superOk* from ClassicJava.

Definition 4 (Well-Formed Super Calls)

$\mathit{superOk}(e) \Longleftrightarrow$ *For all subexpressions* $\mathsf{super}\underline{\ \equiv e_0 : c}.\mathit{md}(e_1 \ldots e_n)$ *of* e, *either* $e_0 = \mathsf{this}$ *or there exists an object reference* obj *such that* $e_0 = \mathit{obj}$.

As with ClassicJava, we must also extend our typing relation to elaborated expressions and situations that may only arise at run-time. The extension mimics ClassicJava; the extended judgment is written $P, \Gamma \vdash_{\underline{\mathsf{e}}} e\ :\ t$, and we elide the precise definition for space reasons. With this last definition, we can state a subject reduction lemma for Jacques; the proof is provided as an appendix.

Lemma 3 (3: Subject Reduction) *If* $P, \Gamma \vdash_{\underline{\mathsf{e}}} e\ :\ t$ *and* $P, \Gamma \vdash_\sigma \mathcal{S}$ *and* $\mathit{superOk}(e)$ *and* $P \vdash \langle e, \mathcal{S}\rangle \hookrightarrow \langle e', \mathcal{S}'\rangle$, *then either* e' *is an error configuration or there exists a type environment* Γ' *such that*

1. $P, \Gamma' \vdash_{\underline{\mathsf{e}}} e'\ :\ t$, *and*

2. $P, \Gamma' \vdash_\sigma \mathcal{S}'$, *and*

3. $\mathit{superOk}(e')$.

To finish the proof of the soundness theorem, we must state and prove a Progress Lemma. For this proof, we need the definition of a redex, given in figure 15, and a unique decomposition lemma.

Lemma 4 (4: Unique Decomposition) *For any closed expression* e *that is not a value, there must exist a context* E *and a redex* e_r *such that* $\mathsf{E}[e_r] = e$. *Further, if* $\mathsf{E}[e_r] = e$ *and* $\mathsf{E}'[e'_r] = e$, *then* $\mathsf{E} = \mathsf{E}'$ *and* $e_r = e'_r$.

The proof is a straightforward structural induction on the term e.

Lemma 5 (5: Progress) *If* e *is closed,* $P, \Gamma \vdash_{\underline{\mathsf{e}}} e\ :\ t$, $P, \Gamma \vdash_\sigma \mathcal{S}$, *and* $\mathit{superOk}(e)$, *then either* e *is a value or there exist* e', $\mathcal{S}'$ *such that* $P \vdash \langle e, \mathcal{S}\rangle \hookrightarrow \langle e', \mathcal{S}'\rangle$, *and* e' *is closed.*

Proof 3 (of lemma 5) The case where e is a value follows directly, so assume that e is not a value. By lemma 4, we know that there exist E and e_r such that $e = \mathsf{E}[e_r]$. The proof proceeds by case analysis on e_r. Most of the cases are as in ClassicJava; we show only those cases that differ here.

- $e_r = v.\mathit{fd}$. If $v = \textsf{null}$, then [*nget*] applies. Otherwise, we know from [FieldRef] that $P, \Gamma \vdash_{\underline{\mathsf{e}}} v : t'$ and $\langle t, \mathit{fd}, s\rangle \in_P t'$. Let $\langle c, \mathit{ctnr}, \mathcal{F}\rangle = \mathcal{S}(v)$. If $s = \textsf{normal}$ or $s = \textsf{contained}$, then by (*esc-5*), $\mathit{fd} \in \mathrm{dom}(\mathcal{F})$, and [*get*] applies. Otherwise, $s = \textsf{acquired}$, and we must evaluate $\mathit{getAcqField}_P(\mathcal{S}, v, \mathit{fd})$. There are three possibilities:

    1. $\mathit{getAcqField}_P(\mathcal{S}, v, \mathit{fd}) = v'$, so [*aget*] applies, and $P \vdash \langle \mathsf{E}[v.\mathit{fd}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v'], \mathcal{S}\rangle$. We know that $\mathsf{E}[v.\mathit{fd}]$ is closed, and substituting v' for $v.\mathit{fd}$ neither removes binding forms nor adds variable references, so $\mathsf{E}[v']$ is closed as required.
    2. $\mathit{getAcqField}_P(\mathcal{S}, v, \mathit{fd}) = \textsf{notFound}$, and [*xaget*] applies.
    3. $\mathit{getAcqField}_P(\mathcal{S}, v, \mathit{fd}) \Uparrow$.

  However, this last case cannot happen. The function $\mathit{getAcqField}_P$ traverses $G_{\mathcal{S}}$, starting from $\mathcal{S}(v)$, and stops either if there are no more outgoing edges or if it finds the desired field. The definition of $G_{\mathcal{S}}$ implies that each node has at most one outgoing edge, so there is exactly one path in the graph that starts with v. And from (*esc-3*), we know that this path is of finite length, so $\mathit{getAcqField}_P$ always terminates, and case 3 is impossible.

- $e_r = v_1.\mathit{fd} = v_2$. This case proceeds much as the previous case; we highlight the differences. First, by [FieldSet], we know that *fd* is in the appropriate class and is not an acquired field. The only interesting case occurs when *fd* is a contained field. If $v_2 = \textsf{null}$, then [*n-ct-set*] applies. Otherwise, there are three interesting cases:

    1. $\mathit{ctnr}(\mathcal{S}(v2)) \neq \textsf{null}$; [*x-ct-set*] applies, and the result of the reduction is closed by the same logic as in the previous case.
    2. $\mathit{ctnr}(\mathcal{S}(v2)) = \textsf{null}$ and $\mathit{canReach}_P(\mathcal{S}, v_1, v_2)$; [*cycle-set*] applies.
    3. $\mathit{ctnr}(\mathcal{S}(v2)) = \textsf{null}$ and $\neg\mathit{canReach}_P(\mathcal{S}, v_1, v_2)$; [*ct-set*] applies.

  The $\mathit{canReach}_P$ predicate also traverses a path in $G_{\mathcal{S}}$, so it must converge to a value by the same logic as used above with $\mathit{getAcqField}_P$.

- $e_r = v.\mathit{md}(v_1 \ldots v_n)$. Parallel to the field reference case above; one of [*call*], [*acall*], [*xacall*], or [*ncall*] applies. If [*call*] or [*acall*] applies, then this reduction replaces the original method call expression with $e_b[v/\textsf{this}, v_1/\mathit{var}_1, \ldots, v_n/\mathit{var}_n]$, where e_b is the body of the method in question and $\mathit{var}_1, \ldots, \mathit{var}_n$ are the method's formal parameters. The expression e_b was originally typechecked, by the [Method] rule, in a type environment Γ, where $\mathrm{dom}(\Gamma) = \{\textsf{this}, \mathit{var}_1, \ldots, \mathit{var}_n\}$. Therefore the free variables of e_b must be a subset of $\mathrm{dom}(\Gamma)$, and so the result of the substitution must be closed, and therefore e' is closed as well.

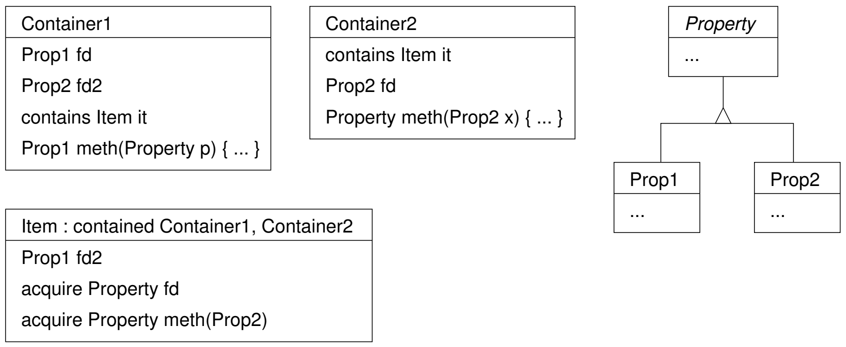
- $e_r = \textsf{super}\underline{\equiv v : c}.\mathit{md}(v_1 \ldots v_n)$. Similar to the previous case, except that $\mathit{superOk}(e)$ implies that $v \neq \textsf{null}$, so one of [*super*], [*asuper*], or [*xasuper*] applies. ■

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 fragment in figure 16 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 simple recursive list implementation, as in figure 17. In this case, a Node instance must be allowed to contain



```
class Main {
    Container1 aCtnr;
    Container2 aCtnr2;
    Item anItem;
    ...
    Object run () {
        ...
        aCtnr.it = anItem;          // anItem contained in aCtnr
        ...
    }
    ...
}
```

Figure 16: Jacques running example

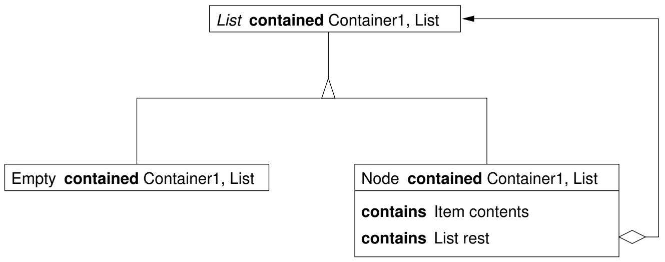

List **contained** Container1, List
Empty **contained** Container1, List
Node **contained** Container1, List
contains Item contents
contains List rest

Figure 17: Recursive lists and acquisition

instances of *List* and particularly other Nodes. Otherwise, acquisition is stopped at the first Node instance, and elements of the list cannot acquire anything from Container1.

In general, forbidding this kind of cycle precludes the use of acquisition in *any* inductively-defined data structure or instance of the composite pattern, severely restricting acquisition's usefulness. Therefore, we must 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*] reduction enforces this restriction at runtime.

This design choice is orthogonal to type safety. Our soundness proof in section 4.4 makes no assumptions about whether G_P contains cycles.

5.2 Acquisition by Value and by Name

In their original description of acquisition, Gil and Lorenz do not address field assignment and its interaction with acquisition. With assignment, it becomes important to determine when acquisition takes place. We considered two options, dubbed "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 16) 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 with acquisition-by-value 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 16, 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 allows just this sort of type variance through the $canAcq_P$ predicate.[4]

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 acquisition 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, as references to aCtnr.fd would reduce to a value of type Prop2. 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 assignments to acquired fields. In our running example, an assignment to anItem.fd would be disallowed because this field is acquired from another object, but assignments to aCtnr.fd would be legal. That aCtnr.fd is acquired by another object is not relevant.

The second option has several drawbacks. First, as long as we preserve subsumption throughout the rest of the language, this restriction could only be checked dynamically, 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 software projects, though, we cannot answer this question definitively, and we need to revisit this issue once we have gained more experience.

[4]At first glance, this situation appears similar to, though not exactly the same as, the traditional notion of contravariance: we have Prop1 $\leqslant_P$ Property but Item $\nleqslant_P$ Container1, so the idea of one type getting larger while the other gets smaller is not applicable.

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.

Allowing programs, however, to move an object from one container to another in a single assignment statement, can violate the invariant above. If, in our running example, we execute the assignment aCtnr2.it = anItem, then anItem's container reference will point to aCtnr2, even though aCtnr still contains anItem.

For this reason, we prohibit programs from moving an object from one container to another in a single assignment statement. Instead, we require the programmer to remove the object from its container before placing it in the new one. In our example, the programmer must first execute aCtnr.it = null, thus setting anItem's container reference to null, before adding anItem to aCtnr2. 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 [13], specifically a switch-container operation.

5.7 Handling incomplete environments

As described above in the discussion of theorem 2, JACQUES cannot statically ensure that references to anItem.fd are well-defined, because anItem may not be in a container that provides the field fd. This arises because anItem is placed into its container by assigning it to the field aCtnr.it, which is only possible after anItem is instantiated. The JACQUES type system does not ensure that code executed between the instantiation and the assignment does not refer to any acquired fields.

An alternative strategy for constructing the object containment graph that would avoid this problem is to require the programmer to supply aCtnr as an argument to anItem's constructor.[5] To prevent the programmer from removing anItem from aCtnr later, we would also disallow assignments to contains fields and instead require a switch-container primitive operation that moves an object from one container to another. Of course, to ensure statically that anItem is always in a container, we would also have to ensure, statically, that the constructor argument and the operands to switch-container are not null, by using a type system similar to that described by Fähndrich [4].

Unfortunately, using such a type system has a number of drawbacks. First, object construction leads to significant complications, as Fähndrich and Leino discuss, because all object fields are initialized to null, even those that are declared to have non-null types. Second, the authors do not discuss how the non-null type annotations "poison" the rest of the program, much as const declarations in C++ can require changing large numbers of other, only tangentially related, classes. Third, requiring a non-null reference as a constructor argument to anItem requires programmers to construct all containment hierarchies by beginning with the root and moving towards the leaves. Experience with an early implementation of JACQUES suggests that this requirement can be extremely awkward in cases where another construction mechanism might be more appropriate. In particular, it disallows the construction pattern commonly used in functional languages, which builds the leaves first and constructs the hierarchy from there.

Because of these drawbacks, we believe that supplying the container as a constructor argument and requiring it to be non-null is a bigger problem, practically speaking, than the possibility of "incomplete context" exceptions. Clearly, we require additional practical experience with acquisition to decide this issue.

5.8 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 $\nleqslant_P$ 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.

[5]David Lorenz proposed this alternative in a private conversation.

By type rule [Meth], 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.

5.9 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 that objects are always in a suitable environment 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 useful, we may add them to the language later.

6 Scaling to Full Java

Jacques is a model of the core of a sequential OO language with acquisition; we have deliberately omitted certain features of full Java [11] and C# [3] to keep the presentation and the proofs simple. In order to extend acquisition to one of these languages, we have to consider how it interacts with other language features not in our model, and we have to explore how it interacts with practical implementation issues.

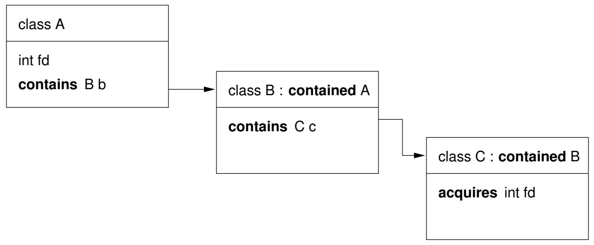
6.1 Acquisition and Java Features

To scale Jacques's design to full Java or C#, 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 might 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.

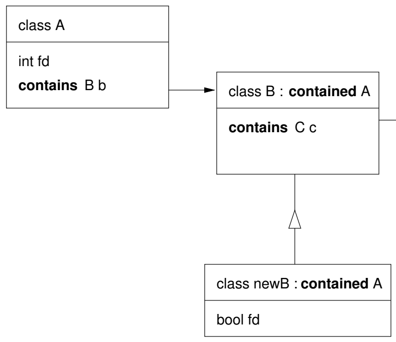
Java-style field shadowing presents a more complicated challenge. In the case where an object acquires a shadowed or shadowing field from its container, it is no longer clear which field should be visible to the acquiring object. Consider the following class definitions:

```
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. The language should, however, provide the programmer with a way to override 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.



The original program



class A
int fd
contains B b
class B : **contained** A
contains C c
class C : **contained** B
acquires int fd
class newB : **contained** A
bool fd

The augmented program

Figure 18: Subclassing, acquisition, and incremental compilation

6.2 Incremental Compilation

By "incremental compilation," we refer to the ability to add a class to a pre-existing program that has already been compiled (and thus type-checked) without needing to recompile the classes from the original program. For environmental acquisition to support this, we must ensure that adding a new class cannot violate invariants that have already been verified on the existing classes. Acquisition makes this challenging, particularly it requires the type checker to verify certain properties of all classes that can contain other objects, plus all of their subclasses. The following example demonstrates the problem.

In the original program (figure 18, top), the [DEFN] rule uses $canAcq_P$ to ensure that C's acquisition of the field fd is safe; in part, this requires that B has no subclasses that define a field fd with an incompatible type. In the augmented program (figure 18, bottom), the new class newB violates this invariant. This violation, however, is not detected when applying the [DEFN] rule to the newB class. Therefore, a naïve solution would have to repeat the type-checking on all classes that could be contained in newB instances, directly or indirectly. In the worst case, this would require type-checking the entire program again.

We can solve this problem by modifying the type checker to annotate each class on a containment path with information regarding the fields and methods whose definitions are constrained. In the example above, the type checker would annotate the class B with the field constraint int fd. Then, when it checks the new class newB, the type checker would proceed up the inheritance hierarchy, verifying that newB does not include any conflicting field and method definitions.

Finally, adding a new class as part of a containment path cannot violate any acquisition invariants in the original program—for the simple reason that a programmer cannot add a single class that is directly part of a containment path. If there is a containment relationship between two classes, the containing class must name the contained class and vice versa. Therefore, adding either class alone causes type-checking to fail with an undefined-class error. As a result, the programmer must add all of the classes in a single containment path in a single increment, at which point the type checker can ensure the necessary invariants for all classes on the path.

7 Related Work

The recent work in ownership types [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. Even though 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 this 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.

APPENDIX

Lemma 6 (3: Subject Reduction) *If* $P, \Gamma \vdash_{\underline{\mathsf{e}}} e \ : \ t$ *and* $P, \Gamma \vdash_{\sigma} \mathcal{S}$ *and* $superOk(e)$ *and* $P \vdash \langle e, \mathcal{S} \rangle \hookrightarrow \langle e', \mathcal{S}' \rangle$*, then either* e' *is an error configuration or there exists a type environment* Γ' *such that*

1. $P, \Gamma' \vdash_{\underline{\mathsf{e}}} e' \ : \ t$*, and*
2. $P, \Gamma' \vdash_{\sigma} \mathcal{S}'$*, and*
3. $superOk(e')$.

Proof 4 (of lemma 3) Since e reduces, there must exist a context E and a redex e_r such that $e = \mathsf{E}[e_r]$. The proof proceeds by structural induction on E; the base case requires a case analysis on the reduction $P \vdash \langle e_r, \mathcal{S} \rangle \hookrightarrow \langle e_r', \mathcal{S}' \rangle$. Most of the details are as in the CLASSICJAVA soundness proof; we present only the interesting cases here.

Reduction [*new*]: $P \vdash \langle \textsf{new}\ c, \mathcal{S}\rangle \hookrightarrow \langle obj, \mathcal{S}'\rangle$, where $obj \notin \mathrm{dom}(\mathcal{S})$, $\mathcal{S}' = \mathcal{S}[obj \mapsto \langle c, \textsf{null}, \mathcal{F}\rangle]$, and $\mathcal{F} = \{fd \mapsto \textsf{null} \mid \langle\_, fd, s\rangle \in_P c \wedge s \neq \textsf{acquired}\}$. This proof case is important because it establishes most of the store invariants.

Let $\Gamma' = \Gamma[obj \mapsto c]$. We have $P, \Gamma \vdash_{\underline{e}} \textsf{new}\ c\ :\ c$, and, by construction, $P, \Gamma' \vdash_{\underline{e}} obj\ :\ c$.

We must also show that $P, \Gamma' \vdash_\sigma \mathcal{S}'$, one clause at a time. We show only the interesting cases.

(*esc-1*) For any object $obj' \in \mathrm{dom}(\Gamma')$, either $obj' = obj$ or $obj' \in \mathrm{dom}(\Gamma)$. In either case, $obj' \in \mathrm{dom}(\mathcal{S}')$.

(*esc-2*) By construction, $\mathrm{dom}(\mathcal{S}') = \mathrm{dom}(\mathcal{S}) \cup \{obj\}$, and $\mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma) \cup \{obj\}$. Since $\mathrm{dom}(\mathcal{S}) \subseteq \mathrm{dom}(\Gamma)$, $\mathrm{dom}(\mathcal{S}') \subseteq \mathrm{dom}(\Gamma')$ as required.

(*esc-3*) $G_{\mathcal{S}'}$ adds one node and no edges to $G_{\mathcal{S}}$, which is itself finite and acyclic. Therefore $G_{\mathcal{S}'}$ must also be finite and acyclic.

(*esc-4*) Consider an object $obj' \in \mathrm{dom}(\mathcal{S}')$ such that $obj' \neq obj$ and $\mathcal{S}'(obj') = \langle c', p', \mathcal{F}'\rangle$. By [*new*], $\mathcal{S}(obj') = \langle c', p', \mathcal{F}'\rangle$, which implies that $\Gamma(obj') = c'$. Then $\Gamma'(obj') = c'$, as required. For obj, we have $\mathcal{S}'(obj) = \langle c, \textsf{null}, \mathcal{F}\rangle$ and $\Gamma'(obj) = c$ by construction, so the necessary relationship holds.

(*esc-5*) For obj, $\mathcal{F}$ has the correct domain by construction. For all other $obj' \in \mathrm{dom}(\mathcal{S}')$, $obj' \in \mathrm{dom}(\mathcal{S})$ so each such object's field function has the correct domain in the old store. Since this reduction does not update any object except obj, the field functions retain the correct domains.

(*esc-7*) This clause remains true for all objects in $\mathrm{dom}(\mathcal{S})$, as this rule does not modify any of their fields. This condition is trivially true for obj, as all of its fields are $\textsf{null}$.

(*esc-9*) $\Longrightarrow$: let obj' be an object in $\mathrm{dom}(\mathcal{S}')$ such that $\mathcal{S}'(obj') = \langle\_, ctnr, \_\rangle$ and $ctnr' \neq \textsf{null}$; as a result of this condition, $obj' \neq obj$. Since this reduction does not change any container references, $\mathcal{S}(obj') = \mathcal{S}'(obj')$. By (*esc-13*), $ctnr \in \mathrm{dom}(\mathcal{S})$, and by (*esc-9*), $P, \mathcal{S} \vdash ctnr \sqsupseteq obj'$. Since this reduction does not modify any fields in existing objects, $P, \mathcal{S}' \vdash ctnr \sqsupseteq obj'$, as required.

$\Longleftarrow$: Let obj' be an object in $\mathrm{dom}(\mathcal{S}')$ such that there exists a $ctnr' \in \mathrm{dom}(\mathcal{S}')$ such that $P, \mathcal{S}' \vdash ctnr' \sqsupseteq obj'$. Since all of obj's fields are $\textsf{null}$, $ctnr' \neq obj$. Because this reduction does not modify any fields in existing objects, we must have $P, \mathcal{S} \vdash ctnr' \sqsupseteq obj'$. Therefore, by (*esc-9*), $\mathcal{S}(obj') = \langle\_, ctnr', \_\rangle$. Since this reduction does not modify any container references, $\mathcal{S}'(obj') = \langle\_, ctnr', \_\rangle$, as required.

(*esc-10*) As before, this clause remains true for all objects in $\mathrm{dom}(\mathcal{S})$ because this reduction rule does not modify any container references. Since obj does not have a container, this condition is trivially satisfied.

Finally, obj has no $\textsf{super}$ subexpressions, so $superOk(obj)$ holds trivially.

Reduction [*aget*]: $P \vdash \langle obj.fd, \mathcal{S}\rangle \hookrightarrow \langle getAcqField_P(\mathcal{S}, obj, fd), \mathcal{S}\rangle$, where $\mathcal{S}(obj) = \langle c, p, \mathcal{F}\rangle$ and $\langle t_{fd}, fd, \textsf{acquired}\rangle \in_P c$ and $getAcqField_P(\mathcal{S}, obj, fd) \neq \textsf{notFound}$.

Let $v = getAcqField_P(\mathcal{S}, obj, fd)$, and let $\Gamma' = \Gamma$.

Since P is well-typed, [Defn] requires that $canAcq_P(c, \langle t_{fd}, fd, \textsf{acquired}\rangle)$ be true. Since

$$getAcqField_P(\mathcal{S}, obj, fd) \neq \textsf{notFound}$$

by assumption, lemma 2 holds, and either $v = \textsf{null}$ or $\mathcal{S}(v) = \langle c_v, \_, \_\rangle$, where $c_v <:_P t$. If $v = \textsf{null}$, then $P, \Gamma' \vdash_{\underline{e}} v\ :\ t$ trivially. Otherwise, we have $P, \Gamma' \vdash_{\underline{e}} v\ :\ t$ by (*esc-4*).

The reduction does not change either Γ or $\mathcal{S}$, so environment-store consistency is trivially preserved. In addition, the reduction does not perform any substitutions, so $superOk(e')$ trivially follows.

Reduction [*ct-set*]: $P \vdash \langle obj.fd = v, \mathcal{S}\rangle \hookrightarrow \langle v, updateCtdField(\mathcal{S}, obj, fd, v)\rangle$, where $v \in \mathrm{dom}(\mathcal{S})$ and $\langle\_, fd, \textsf{contained}\rangle \in_P classTag(\mathcal{S}(obj))$ and $ctnr(\mathcal{S}(v)) = \textsf{null}$ and $\neg canReach_P(\mathcal{S}, obj, v)$.

Let

$$\begin{aligned}
\Gamma' &= \Gamma\\
\mathcal{S}' &= updateCtdField(\mathcal{S}, obj, fd, v)\\
\mathcal{F} &= fields(\mathcal{S}(obj))\\
\mathcal{F}' &= fields(\mathcal{S}'(obj)).
\end{aligned}$$

In particular, $\mathcal{S}$ and $\mathcal{F}$ are the store and *obj*'s fields before the reduction, and $\mathcal{S}'$ and $\mathcal{F}'$ are the store and the fields after the reduction. Therefore, $\mathcal{F}(\mathit{fd})$ is the value of the field before the assignment, and v is its value afterwards. Note that $\mathcal{F}(\mathit{fd}) \neq \mathit{obj}$ by (*esc-3*), and $v \neq \mathit{obj}$, or $\mathit{canReach}_P(\mathcal{S}, \mathit{obj}, v)$ would be true.

We have $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathit{obj}.\mathit{fd} = v \ : \ t$, so $P, \Gamma \vdash_{\underline{\mathsf{e}}} v \ : \ t$ by [FieldSet]. By (*esc-4*), then, $\mathit{classTag}(\mathcal{S}(v)) = c$ where $c <:_P t$. Since *updateCtdField* does not modify any object's class tag, $\mathit{classTag}(\mathcal{S}'(v)) = c$, so we have $P, \Gamma' \vdash_{\underline{\mathsf{e}}} v \ : \ t$ as required.

Now ensure environment-store consistency. By the definition of *updateCtdField*, $\mathrm{dom}(\mathcal{S}') = \mathrm{dom}(\mathcal{S})$ and $\mathrm{dom}(\mathcal{F}) = \mathrm{dom}(\mathcal{F}')$. (Again, we omit the straightforward cases for space reasons.)

(*esc-3*) This reduction creates $G_{\mathcal{S}'}$ from $G_{\mathcal{S}}$ through two operations. First, if $\mathcal{F}(\mathit{fd}) \neq \mathsf{null}$, then it removes the edge from $\mathcal{F}(\mathit{fd})$ to *obj*, which cannot create any cycles. Then, it adds an edge from v to *obj*. Since $G_{\mathcal{S}}$ has no cycles, this new edge can create a cycle only when there is a path from *obj* to v in $G_{\mathcal{S}}$. But since $\mathit{canReach}_P(\mathcal{S}, \mathit{obj}, v)$ is false, this is impossible, and therefore $G_{\mathcal{S}'}$ has no cycles. Finally, $G_{\mathcal{S}'}$ and $G_{\mathcal{S}}$ have the same number of nodes, so $G_{\mathcal{S}'}$ is finite.

(*esc-6*) This reduction only modifies the fields of *obj*; this property is preserved for all other objects. By construction, $\mathrm{rng}(\mathcal{F}') \subseteq \mathrm{rng}(\mathcal{F}) \cup \{v\}$. Since $v \in \mathrm{dom}(\mathcal{S})$, $\mathrm{rng}(\mathcal{F}) \cup \{v\} \subseteq \mathrm{dom}(\mathcal{S}) \cup \{\mathsf{null}\}$. Since $\mathrm{dom}(\mathcal{S}') = \mathrm{dom}(\mathcal{S})$, we have $\mathrm{rng}(\mathcal{F}') \subseteq \mathrm{dom}(\mathcal{S}') \cup \{\mathsf{null}\}$ as required.

(*esc-7*) The only interesting case is the field *fd* within *obj*; no other fields are modified. From $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathit{obj}.\mathit{fd} = v \ : \ t$ and [FieldSet], we have $\langle t, \mathit{fd}, \_ \rangle \in_P c$, where $\Gamma(\mathit{obj}) = c$ and $\mathcal{S}(\mathit{obj}) = \langle c, p, \mathcal{F} \rangle$. Further, $P, \Gamma \vdash_{\underline{\mathsf{e}}} v \ : \ t$. This satisfies the clause.

(*esc-8*) This clause holds true for all objects other than *obj* and for all fields of *obj* other than *fd*. For this case, assume that there exists a field $\mathit{fd}' \neq \mathit{fd}$ such that $\mathcal{F}(\mathit{fd}') = v$, which implies that $\mathcal{F}(\mathit{fd}') = v$ as well. Then $P, \mathcal{S} \vdash \mathit{obj} \sqsupseteq v$ and by (*esc-12*), $\mathit{ctnr}(\mathcal{S}(v)) = \mathit{obj}$. But this violates one of the conditions for this reduction, so this clause's antecedents cannot be true and the clause is thus trivially satisfied.

(*esc-9*) $\Longrightarrow$: Let obj' be an object in $\mathrm{dom}(\mathcal{S}')$ such that $\mathcal{S}'(\mathit{obj}') = \langle \_, \mathit{ctnr}, \_ \rangle$ and $\mathit{ctnr} \neq \mathsf{null}$. Case analysis on obj':

- $\mathit{obj}' = \mathcal{F}(\mathit{fd})$. Cannot happen because $\mathcal{S}'(\mathcal{F}(\mathit{fd})) = \langle \_, \mathsf{null}, \_ \rangle$.
- $\mathit{obj}' = \mathit{obj}$, the object whose field is updated. Let $\langle \_, p, \_ \rangle = \mathcal{S}'(\mathit{obj}')$. Because $G_{\mathcal{S}}$ is acyclic, $\mathit{obj}' \neq \mathcal{F}(\mathit{fd})$, and because $\neg\mathit{canReach}_P(\mathcal{S}, \mathit{obj}', v)$, $\mathit{obj}' \neq v$. Therefore obj' is not one of the two objects whose containers are modified by this reduction, and $\mathit{ctnr}(\mathcal{S}'(\mathit{obj}')) = \mathit{ctnr}(\mathcal{S}(\mathit{obj}')) = p$. Therefore, we have $P, \mathcal{S} \vdash p \sqsupseteq \mathit{obj}'$. Now, since $p \neq \mathit{obj}'$ (again because $G_{\mathcal{S}}$ is acyclic), this reduction does not modify any of p's fields, so we must have $P, \mathcal{S}' \vdash p \sqsupseteq \mathit{obj}'$ as required.
- $\mathit{obj}' = v$, the new value of the field. By [*ct-set*], $\mathit{ctnr} = \mathit{obj}$ and $\mathcal{F}'(\mathit{fd}) = v$, so $P, \mathcal{S}' \vdash \mathit{obj} \sqsupseteq v$ and thus $P, \mathcal{S}' \vdash \mathit{obj} \sqsupseteq \mathit{obj}'$ as required.
- obj' is none of $\mathcal{F}(\mathit{fd})$, *obj*, or v. Since this reduction only changes the container references of $\mathcal{F}(\mathit{fd})$ and v, we must have $\mathit{ctnr}(\mathcal{S}(\mathit{obj}')) = \mathit{ctnr}(\mathcal{S}'(\mathit{obj}'))$. Therefore, $P, \mathcal{S} \vdash \mathit{ctnr} \sqsupseteq \mathit{obj}'$. If $\mathit{ctnr} \neq \mathit{obj}$, then *ctnr*'s fields are not modified, so $P, \mathcal{S}' \vdash \mathit{ctnr} \sqsupseteq \mathit{obj}'$. Otherwise, if $\mathit{ctnr} = \mathit{obj}$, then there must exist some field $\mathit{fd}' \neq \mathit{fd}$ such that $\mathcal{F}(\mathit{fd}') = \mathit{obj}'$, where fd' is a `contained` field. Since this reduction only changes the field *fd*, $\mathcal{F}'(\mathit{fd}') = \mathit{obj}'$, and thus $P, \mathcal{S}' \vdash \mathit{ctnr} \sqsupseteq \mathit{obj}'$ as required.

$\Longleftarrow$: Let obj_1 and obj_2 be arbitrary objects in $\mathrm{dom}(\mathcal{S}')$ such that $P, \mathcal{S}' \vdash \mathit{obj}_1 \sqsupseteq \mathit{obj}_2$. Case analysis on obj_1:

- $\mathit{obj}_1 = \mathcal{F}(\mathit{fd})$. Then $\mathit{obj}_1 \neq \mathit{obj}$, or $G_{\mathcal{S}}$ would have a cycle. Since this reduction does not modify obj_1's fields, $P, \mathcal{S} \vdash \mathit{obj}_1 \sqsupseteq \mathit{obj}_2$ and therefore $\mathit{ctnr}(\mathcal{S}(\mathit{obj}_2)) = \mathit{obj}_1$. Further, $\mathit{obj}_2 \neq \mathcal{F}(\mathit{fd})$; otherwise (because this reduction does not modify $\mathcal{F}(\mathit{fd})$'s fields) $G_{\mathcal{S}}$ would have had a cycle. Similarly, $\mathit{obj}_2 \neq v$; otherwise we'd have $\mathit{ctnr}(\mathcal{S}(v)) = \mathit{obj}_1$, but we know that $\mathit{ctnr}(\mathcal{S}(v)) = \mathsf{null}$. Therefore, this reduction does not modify obj_2's container pointer, and $\mathit{ctnr}(\mathcal{S}'(\mathit{obj}_2)) = \mathit{obj}_1$, as required.

- $obj_1 = obj$. By this reduction and (*esc-8*), we know that $\neg(P, \mathcal{S}' \vdash obj \sqsupseteq \mathcal{F}(fd))$, so $obj_2 \neq \mathcal{F}(fd)$. Further, $obj_2 \neq obj$; otherwise $G_{\mathcal{S}'}$ would have a cycle, and we previously proved that impossible. If $obj_2 = v$, then this reduction ensures that $ctnr(\mathcal{S}'(v)) = obj$, as required. Otherwise, since we do not modify any of obj's other fields, we must have $P, \mathcal{S} \vdash obj_1 \sqsupseteq obj_2$, which implies that $ctnr(\mathcal{S}(obj_2)) = obj_1$. Since this reduction does not modify obj_2's parent pointer, $ctnr(\mathcal{S}'(obj_2)) = obj_1$, as required.
- $obj_1 = v$. This reduction does not modify obj_1's fields, so we must have $P, \mathcal{S} \vdash obj_1 \sqsupseteq obj_2$, and therefore that $ctnr(\mathcal{S}(obj_2)) = obj_1$. Now, because $P, \mathcal{S} \vdash obj \sqsupseteq \mathcal{F}(fd)$, we know that $ctnr(\mathcal{S}(\mathcal{F}(fd))) = obj$. Since $obj_1 \neq obj$, $obj_2 \neq \mathcal{F}(fd)$. And $obj_2 \neq v$, or $G_{\mathcal{S}}$ would have a cycle. Therefore, this reduction does not change obj_2's parent pointer, so $ctnr(\mathcal{S}'(obj_2)) = obj_1$ as required.
- Finally, $obj_1 \notin \{\mathcal{F}(fd), obj, v\}$. Since this reduction does not modify obj_1's fields, we have $P, \mathcal{S} \vdash obj_1 \sqsupseteq obj_2$ and therefore $ctnr(\mathcal{S}(obj_2)) = obj_1$. Because $obj_1 \neq obj$, $obj_2 \neq \mathcal{F}(fd)$. And because $ctnr(\mathcal{S}(v)) = \mathsf{null}$, $obj_2 \neq v$. Therefore, this reduction does not modify obj_2's parent pointer, and $ctnr(\mathcal{S}'(obj_2)) = obj_1$ as required.

(*esc-10*) This reduction changes the parent pointers of exactly two objects, $\mathcal{F}(fd)$ and v; the property is trivially preserved for all others. By the definition of the reduction, $ctnr(\mathcal{S}'(\mathcal{F}(fd))) = \mathsf{null}$, so the requirement for the first object is trivially satisfied. For the second object, $ctnr(\mathcal{S}'(v)) = obj$. Let $c_1 = classTag(\mathcal{S}'(obj))$ and $c_2 = classTag(\mathcal{S}'(v))$. By [*ct-set*], we know that $\langle t, fd, \mathsf{contained}\rangle \in_P c_1$; by [FieldSet], $c_2 \leqslant_P t$. By [Defn], $t \sqsubset_P c_1$, and also $containers_P(c_2) \supseteq containers_P(t)$. Therefore, $c_2 \sqsubset_P c_1$ as required.

(*esc-11*) The only objects whose parent pointers are modified by this reduction are v and $\mathcal{F}(fd)$. By construction, we have $ctnr(\mathcal{S}'(v)) = obj \in \mathrm{dom}(\mathcal{S}')$, and $ctnr(\mathcal{S}'(\mathcal{F}(fd))) = \mathsf{null}$.

Therefore, $P, \Gamma' \vdash_\sigma \mathcal{S}'$.

Finally, since the result of this reduction is the value v, it cannot contain any super expressions, and $superOk(v)$ is true.

Reduction [*acall*]: We have

$$\begin{aligned} &P \vdash \langle obj.md(v_1 \ldots v_n), \mathcal{S}\rangle \hookrightarrow \langle e_b[obj'/\mathsf{this}, v_1/var_1, \ldots, v_n/var_n], \mathcal{S}\rangle,\\ &\quad \text{where } \langle md, (t_1 \ldots t_n \to t_{md}), \_, e, \mathsf{acquired}\rangle \in_P classTag(\mathcal{S}(obj))\\ &\quad \text{and } \langle obj', (var_1, \ldots, var_n), e_b\rangle = getAcqMethod_P(\mathcal{S}, obj, md). \end{aligned}$$

Take $\Gamma' = \Gamma$. Since P is well-typed, [Defn] requires that

$$canAcqMeth_P(c, \langle md, (t_1 \ldots t_n \to t_{md}), V, e, \mathsf{acquired}\rangle).$$

By [*acall*], $getAcqMethod_P(\mathcal{S}, obj, md) = \langle obj', (var_1, \ldots, var_n), e_b\rangle$. Therefore lemma 2 applies, so

- $\mathcal{S}(obj') = \langle c_v, \_, \_\rangle$, and
- $\langle md, (t'_1 \ldots t'_n \to t'), (var_1, \ldots, var_n), e_b, \_\rangle \in_P c_v$, and
- $(t'_1 \ldots t'_n \to t') <:_P (t_1 \ldots t_n \to t_{md})$.

By [Call], $P, \Gamma \vdash_{\underline{\mathsf{e}}} obj.md(v_1 \ldots v_n) \ : \ t$ implies that $t = t_{md}$ and that

$$P, \Gamma \vdash_{\underline{\mathsf{e}}} v_i \ : \ t_i \text{ for } i \in [1, n].$$

By the definition of subtyping, this implies that $P, \Gamma \vdash_{\underline{\mathsf{e}}} v_i \ : \ t'_i$ for all $i \in [1, n]$. Now, by [Defn] and [Method], $\langle md, (t'_1 \ldots t'_n \to t), (var_1, \ldots, var_n), e_b, \_\rangle \in_P c_v$ requires that

$$P, \Gamma[\mathsf{this} \mapsto c_v, var_1 \mapsto t'_1, \ldots, var_n \mapsto t'_n] \vdash_{\underline{\mathsf{e}}} e_b \ : \ t'.$$

Therefore, by a substitution lemma,[6]

$$P, \Gamma \vdash_{\underline{e}} e_b[obj'/\mathsf{this}, v_1/var_1, \ldots, v_n/var_n] \ : \ t',$$

and by subtyping, $P, \Gamma \vdash_{\underline{e}} e_b[obj'/\mathsf{this}, v_1/var_1, \ldots, v_n/var_n] \ : \ t_{md}$ as required.

This reduction does not change either Γ or $\mathcal{S}$, so environment-store consistency is trivially preserved. It may, however, introduce new super expressions, though all of these introduced expressions come from e_b, and all of these expressions were annotated by the elaboration phase. Therefore, all of the super expressions in e_b are annotated with this. The only part of the substitution that can have any effect on these expressions is $e_b[obj'/\mathsf{this}]$, which replaces the annotation with obj'. Therefore *superOk* holds. ■

2 Acknowledgments

We thank David Lorenz for taking the time to discuss his prior work with us. Mitch Wand provided invaluable assistance with JACQUES's soundness proof. Erik Ernst pointed out an important omission in the earlier, informal versions of $canAcq_P$ and $canAcqMeth_P$ [2]. David Richter suggested the possibility of using type systems for non-nullable pointers, as described in section 5.7. Finally, we thank the POPL 2005 anonymous reviewers for their comments, insights, and suggestions.

References

[1] 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.

[2] Richard Cobbe and Matthias Felleisen. Environmental acquisition revisited. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 14–25, New York, NY, USA, 2005. ACM Press.

[3] ECMA. Standard ECMA-334: C# language specification. Available on the web as `http://www.ecma-international.org/publications/files/ecma-st/ECMA-334.pdf`, December 2002.

[4] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 302–312. ACM Press, 2003.

[5] 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.

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

[7] Jim Fulton. Acquisition. `http://olympus.het.brown.edu/doc/python-extclass/Acquisition.html`, 1998.

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

[9] 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.

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

[6]The substitution lemma takes the standard form; it is omitted here for space reasons.

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

[12] Kathryn E. Gray and Matthew Flatt. ProfessorJ: A gradual intro to Java through language levels. In *OOPSLA Educators' Symposium*, 2003.

[13] 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.

[14] 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.

[15] 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`.

[16] 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.

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

[18] Sun Microsystems. Java foundation classes (JFC/Swing). `http://java.sun.com/products/jfc/index.jsp`, 1998.

[19] Guido van Rossum. Python. `http://www.python.org/`, 1991.

[20] 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.