Towards a Formalization for COM Part I: The Primitive Calculus

Riccardo Pucella
Department of Computer Science
Cornell University
Ithaca, NY 14853
riccardo@cs.cornell.edu

ABSTRACT

We introduce in this paper a typed calculus intended to capture the execution model of COM. The innovation of this calculus is to model very low-level aspects of the COM framework, specifically the notion of interface pointers. This is handled by specifying an allocation semantics for the calculus, thereby modeling heap allocation of interfaces explicitly. Having an explicit way of talking about interface pointers allows us to model in a reasonable way the notions of interface sharing and object identity. We introduce a type system that can be used to disambiguate between specification and implementation of interfaces. The type system moreover can capture a notion of COM conformance, that is, the legality of COM components. We discuss extensions of the calculus to handle subtyping of interfaces, dynamic interface negotiation and aggregation.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Modules, packages*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs; D.3.2 [**Programming Languages**]: Language Classifications—*Applicative languages, Object-oriented languages*

General Terms

Design, Languages

Keywords

COM, components, interfaces, calculus, formalism, type system

1. INTRODUCTION

The *Component Object Model* (COM), introduced by Microsoft in 1995, is an architectural model intended to promote the safety, interoperability, and distribution of largely-independent units of functionality [4, 10]. The driving idea was to provide a simple model that could be used to build more advanced infrastructures for application development. The work on COM emerged from an attempt at isolating the core mechanisms of OLE, the *Object Linking and Embedding* infrastructure used by the Windows operating system to manage document-centric abstractions [1].

The basis of the COM model is the notion of *interface*. One can think of an interface as a view on a given "object", namely the instance of a component. For example, a spell-checking component instance may have a dictionary view that allows it to be accessed as a dictionary. Through this view, one can query the instance for the words it knows. Another view of this component may be a spell-checking view, that can perform the actual spell-checking of a document. What makes the COM framework interesting is that these interfaces are essentially independent of each other! Each interface presents various functions that can be called, and different interfaces for a given component instance need not even share state. However, to maintain the illusion that these interfaces are views of an underlying "object", COM imposes requirements on interfaces, such as the requirement that given any view of an "object", we can get at any of the others views, and so on.

Although one of the goal was to have a simple model, by forcing as little structure as possible, it has proved surprisingly difficult to formalize. Subtleties in the interpretation of the model have recently arisen, pointing to the need for such a formalization. For instance, recent results by Sullivan, Marchukov, and Socha [15] highlight unexpected interactions between two core mechanisms of COM, interface negotiation and aggregation. These interactions invalidate a naive reading of the COM specification. One must realize that these are not even the most complicated protocols in COM, compared with issues such as security, licensing, marshalling, remote objects and especially threading models. This is compounded by the fact that the ideas underlying COM have been adopted by other frameworks, such as the XP-COM framework used in Mozilla, and are being carried over to the .NET framework from Microsoft.

Why is it so hard to reason about COM? The COM specification [4] describes the model in terms of how implementations should behave, and does so by specifying rules that guarantee good behavior. One problem is that the required behavior is described as a mix of specifications and a reference implementation in an object-oriented language such as C++. It is not clear, for example, how much the specifications and rules of usage rely on the reference implementation. Moreover, the specification does not lend itself well to the verification that the rules actually enforce good behavior. As we mentioned, formalizations that have emerged since the COM specification was "published" have already exhibited subtle interactions between the rules [15].

OOPSLA'02, November 4-8, 2002, Seattle, Washington, USA.

In this paper, we introduce a framework to reason about the interaction between the rules and the effect of the rules on programs. This is achieved by capturing the execution model of COM via a primitive typed calculus, λ^{COM}, executing over an abstract machine that models the system at the level of interfaces, through explicit interface pointers. Essentially, we explicitly model allocation of new interfaces in a "heap". This allows us to express notions such as interface identity, required to define the identity of components. The key feature of our calculus is its type system, which statically determines if a component and its set of interfaces is COM conformant. Roughly speaking, a COM component is said to be conformant (or legal) if it satisfies the COM rules of usage. Recent research by Sullivan and Marchukov [14] suggests that the notion of COM conformance cannot be fully captured statically, and one goal of our project is to examine the exact implications of such research.

Various approaches have attempted to formalize the COM framework, including the aforementioned research [14, 15], as well as calculi-based methods such as COMEL [2]. None of these approaches capture the computational model accurately. The concepts underlying COM appear intuitive, but turn out to be subtle. The aim of this project is to attempt to provide a semantic foundation for the study of the component technology underlying COM, from a programming language semantics point of view.

The rest of this paper is structured as follows. We give an overview of COM in Section 2, at the level required to understand our work. In Section 3 we introduce our calculus, λ^{COM}, and the type system enforcing COM conformance. In Section 4, we present the operational semantics of the calculus. In Section 5, we point out natural extensions of our work. We review related work in Section 6, and then conclude with a discussion of our design decisions and future work. The proof of our technical results have been relegated to the full paper.

2. THE COMPONENT OBJECT MODEL

COM is an architectural model based on largely-independent units of functionality called *components*. COM components are binary objects: machine code and data laid out in memory and behaving according to the rules of COM. To provide access to its functionality, a component implements one or more *interfaces*. An *interface* is itself a binary structure, whose layout is specified by the COM framework. Precisely, an interface is a pointer to a table of pointers to functions of the component (the *interface table*). Typically, the functions provided through an interface are semantically related. For instance, they may implement a specific service. The only way for client code to access the code of a component is through one of those interfaces.

Consider the following classical example of a component, one that performs spell-checking duties. Such a component may have two interfaces: *ISpell*, to perform spell-checking, and *IDictionary*, to access the underlying dictionary. The *ISpell* interface would allow one to spell-check a piece of text, while the *IDictionary* interface would contain functions to update the dictionary, such as adding new words, or alternate spellings.

The *type* of an interface is the list of operations it supports, and their order (as implemented in the interface table), along with the types of its operations. An interface can *inherit* from another interface, which simply says that it provides all the functions of that other interface, and maybe additional ones as well. Every interface is required to ultimately inherit from *IUnknown*, which is a special interface in COM. The *IUnknown* interface specifies three functions: *QueryInterface*, *AddRef*, and *Release*. These functions are the core COM mechanisms for interface negotiation and memory management. We will not be concerned with memory management issues in this paper.

Interface negotiation through the *QueryInterface* function is the only way in which one can use components. Recall that due to encapsulation, a component can only be accessed through one of its interfaces. This begs the question of how one gets a hand on an interface to a component. The answer is that one can query a component for an interface, through *QueryInterface*. Since *QueryInterface* is specified by *IUnknown*, from which every interface must inherit, every interface in every component must implement *QueryInterface*. In essence, if one has an interface to a component, one can get another interface to a component. To understand querying, one must understand the naming scheme for interfaces. Until now, we have described interfaces using mnemonic names such as *IDictionary*. In truth, every interface has an associated *interface identifier*, or IID. This IID uniquely identifies the type of the interface. Any interface with a given IID must implement exactly the functions specified by the type associated with the IID. Any new interface not corresponding to an existing interface identifier must be assigned such an identifier by the creator of the interface, as part of the design of the interface. The IID carries some "semantic" meaning. For instance, there may be an IID reserved for interfaces to dictionaries used by spell-checkers, with exactly the same type as provided by another IID reserved for interfaces to dictionaries used by natural language processors. The IID is what one queries for when calling *QueryInterface*. When one queries a component for an interface with the given IID, either the call succeeds and one gets a pointer to the requested interface (i.e., a pointer to a table of function pointers), or the call fails with an indication that such an interface is not available. Note two things: there need not be a unique interface with a given IID implemented by a component; the component is free to choose which interface pointer to return. Second, to query a component for an interface, one needs to know the interface IID one is interested in. COM does not provide any mechanism through which one can ask a component about the interfaces it provides. The intuition is that this information would be useless to a client unless it was programmed to handle such interfaces, in which case it can simply query for them.

In summary, to get an interface of a component, one needs to call *QueryInterface* through another interface of a component. How does one get an initial interface to a component? When one creates a component (on Win32 systems, this is done through a call to the Win32 function *CoCreateInstance*), one specifies the initial interface one wants on the component. Note that every component is required to implement at least one specific interface, namely *IUnknown*, and thus one can always query for it at creation time. Recall that *IUnknown* simply provides a *QueryInterface* and memory management functions. (COM imposes obligations with respect to *IUnknown* in two distinct ways: every interface must inherit from *IUnknown*, and every component must support the *IUnknown* interface directly.)

The *QueryInterface* mechanism is purely local: from a given interface, one can query for other interfaces. To be usable, a component must impose restrictions on the ways in which *QueryInterface* as implemented by the interfaces behaves. The *QueryInterface* functions in a component must have the following properties:

1. **Stability.** If at some point, a *QueryInterface* on some interface for an interface with IID I succeeds, then every subsequent call to *QueryInterface* on that interface requesting an interface with IID I must also succeed. Similarly, if at some point, a *QueryInterface* on some interface for an interface with IID I fails, then every subsequent call to *QueryInterface* on that interface requesting an interface with IID I must fail.

$$
\begin{array}{llll}
\textit{types} & \tau & ::= & \text{int} \mid \tau_1 \to \tau_2 \mid [I, \iota] \mid [\iota] \\
\textit{interface types} & \iota & ::= & \langle I_1{:}\iota_1, \ldots \rangle_t \mid \alpha \mid \mu_i(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n) \\
\textit{tags} & t & ::= & i \mid \bot \\
\\
\textit{declarations} & d & ::= & x \mid \langle I \mid l_1{=}e_1, \ldots \mid I_1{\leftarrow}d_1, \ldots \rangle \mid \textbf{rec}\ x_1{=}d_1, \ldots\ \textbf{in}\ d \\
\textit{values} & v & ::= & i \mid \lambda x{:}\tau.e \mid \ell \mid \textbf{component}\ (\ell) \\
\textit{expressions} & e & ::= & x \mid i \mid \lambda x{:}\tau.e \mid e_1\ e_2 \mid e.l \mid e\#I \mid e\# \mid \\
 & & & \textbf{rec}\ x_1{=}d_1, \ldots\ \textbf{in}\ e \mid \textbf{unroll}_\iota\ (e) \mid \\
 & & & \langle I \mid l_1{=}e_1, \ldots \mid I_1{\leftarrow}e'_1, \ldots \rangle \mid \textbf{component}\ (e)
\end{array}
$$

Figure 1: λ^{COM} syntax

2. **Reflexivity.** A *QueryInterface* from an interface with IID I for an interface with IID I always succeeds.

3. **Symmetry.** If a *QueryInterface* from an interface with IID I for an interface with IID I' succeeds, then a *QueryInterface* on the resulting interface for an interface with IID I must succeed.

4. **Transitivity.** If a *QueryInterface* from an interface with IID I for an interface IID I' succeeds, and a *QueryInterface* on the resulting interface for an interface with IID I'' succeeds, then a *QueryInterface* from the original interface for an interface with IID I'' must also succeed.

These properties say nothing about whether the returned interface pointers are the same. In general, it need not be the case that querying for a given interface from two different interfaces returns the same interface pointer. This flexibility allows components to optimize table layouts, and even to reduce network communication in the case of distributed components.

The only case where the component is required to return the same pointer to an interface is when an interface of the component is queried for *IUnknown*. This leads to the following rule:

5. **Uniqueness.** Querying any interface of a component for *IUnknown* must succeed and always return the same pointer.

This unique *IUnknown* pointer can be used to define a notion of component identity: two interfaces are interfaces of the same component if querying them for *IUnknown* returns the same pointer.

Component composition, that is, creating new components from old, is handled in COM in one of two ways, namely containment and aggregation. The only such way we address in this paper is containment. Containment is straightforward: a component C_1 (called the outer component) is said to *contain* a component C_2 (called the inner component) if C_1 uses C_2 in its implementation. In other words, C_1 is a client of C_2. The only requirement for containment is that upon creation, the outer component should create the inner component. We will not deal with aggregation in this paper, as it introduces complexities of its own. We will discuss aggregation in Section 5.

3. THE λ^{COM} CALCULUS

In this section, we introduce the λ^{COM} calculus, a statically-typed language that models COM at the level of the pointer-based implementation of interfaces. The calculus is a functional calculus based on the call-by-value simply-typed λ-calculus. After presenting the model of the calculus, we explain and motivate the main constructs and typing rules, and discuss how we capture COM conformance.

3.1 The model

The cursory look at COM in Section 2 illustrates the fact that interfaces are the core elements of the COM infrastructure. Our calculus is based on the following very general view of interfaces. Interfaces, as we saw, can be understood as records, in that each interface contains fields that contains values. Each interface is associated an IID, describing the fields it contains. Moreover, each interface is *linked* to other interfaces, where the link is labelled by the IID of the target interface. Intuitively, querying an interface i for an interface with IID I means returning the interface obtained by following the link labelled I. Note that there is no sharing of structure—each interface is its own record, distinct from other interfaces (although of course they can be linked). Hence, we avoid conflating issues of inheritance in the model. Another thing to note is that this approach encodes a particular "implementation" of the *QueryInterface* mechanism. Querying for an interface is done by table lookup. One cannot, within the model, specify that the interface to be returned as a result of a particular *QueryInterface* depends on runtime values. We will see later in this section where this restriction comes in handy. In Section 7, we will revisit this particular assumption.

It goes without saying that different interfaces can link to common interfaces, and that linking can lead to cycles among interfaces. To account for all these aspects, we introduce the notion of a *heap*, where interfaces are stored. Each stored interface gets an *address* and accessing interfaces is done by going through their address. Hence, the label of an interface can be understood as a *pointer* to an interface. Interfaces in this model can be seen as a directed graph structure. The heap is simply a representation of that graph. What is a component instance in such a setting? An instance is a *connected component* (in the graph-theoretic sense) of this graph structure. Given such a description, it is not particularly relevant what an instance is: any interface in the connected component can be seen as a "handle" to the component. Unfortunately, this approach allows us to describe connected components that do not behave the way COM components should behave. Specifically, the rules of behavior of Section 2 are not enforced by this model. For example, querying any interface in a connected component for an *IUnknown* interface need not always yield the same interface as a result, violating the *IUnknown* property of COM components. Therefore, we enforce statically the COM conformance of components in our type system, by identifying a specific interface as the "official entry point" into the component. Not surprisingly, this entry point will be the common *IUnknown* interface required by COM conformance!

3.2 The language

The calculus is an extension of the simply-typed λ-calculus. The

full syntax is given in Figure 1. The types for the values in the calculus are given as follows:

$$\begin{array}{rcl} \tau & ::= & \text{int} \mid \tau_1 \to \tau_2 \mid [I, \iota] \mid [\iota] \\ \iota & ::= & \langle I_1{:}\iota_1, \ldots \rangle_t \mid \alpha \mid \mu_i(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n) \end{array}$$

The types τ comprise the base type int, as well as functional types $\tau_1 \to \tau_2$. The type $[I, \iota]$ denotes the type of an interface. Interfaces are first-class, in that they can be passed to and returned from functions. As we noted above, an interface is essentially a record, and is moreover linked to other interfaces. The type for interfaces reflects that structure. First, we need to assume a set IIDS of interface identifiers, and let $I_1, I_2, \ldots$ range over elements of IIDS. We also assume a function $\mathcal{I}$ taking an interface identifier and returning a record $\{l_1 : \tau_1, \ldots\}$ of field names and corresponding types. This means that we do not need to specify the field types in an interface type if we have the interface identifier corresponding to the interface. This approach captures the fact that interfaces are fixed by IID. The type $[\iota]$ represents the type of a component, that is, intuitively, the type of the interface that is the entry point of a given component. As we will see shortly, this interface must have IID *IUnknown*, and hence we don't need to specify this IID in the type.

An interface basically has type $[I, \langle I_1{:}\iota_1, \ldots \rangle_t]$, meaning it has an IID I, and hence contains fields given by $\mathcal{I}(I)$, and is linked to interfaces with IID $I_1, \ldots, I_n$ of type $\iota_1, \ldots, \iota_n$, respectively. The interface type is also tagged with a tag t, to which we'll return in Section 3.3. (It is used to check an aspect of COM conformance.) A tag is either an integer or the special tag $\perp$. We will omit the tag when it is not relevant. As we alluded to above, the linking between interfaces can lead to cyclic structures. For example, an interface i_1 of type $[I_1, \iota_1]$, that can be queried for an interface with IID I_2 of type ι_2, which itself can be queried for an interface of with IID I_1, of type ι_1. To type such cyclic structures, we introduce the recursive interface type $\mu_i(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n)$. Technically, this construct takes the simultaneous fixed point of $\iota_1, \ldots, \iota_n$ over the variables $\alpha_1, \ldots, \alpha_n$, and gives back the fixed point corresponding to ι_i. Consider the recursive structure given above. If we let α_1 be the interface type of i_1, and α_2 be the interface type corresponding to querying i_1 for IID I_1, then α_1 is $\langle I_2{:}\alpha_2 \rangle_{t_1}$ and α_2 is $\langle I_1{:}\alpha_1 \rangle_{t_2}$. Taking fixed points, the type of i_1 becomes $[I_1, \mu_1(\alpha_1, \alpha_2).(\langle I_2{:}\alpha_2 \rangle_{t_1}, \langle I_1{:}\alpha_1 \rangle_{t_2})]$, and the type of i_2 becomes $[I_2, \mu_2(\alpha_1, \alpha_2).(\langle I_2{:}\alpha_2 \rangle_{t_1}, \langle I_1{:}\alpha_1 \rangle_{t_2})]$, as expected.

Expressions in the language are used to construct values of the appropriate types. As the calculus is based on the λ-calculus, it has the standard constructs for functional abstraction ($\lambda x{:}\tau.e$) and application ($e_1\ e_2$), as well as integer constants.[1] Type judgments assign types to expressions. The basic typing judgment has the form:

$$\mathcal{I}; \Psi; \Gamma \vdash e : \tau$$

where $\mathcal{I}$ is the IID assignment function referred to earlier, Ψ is the type of the heap, an assignment of types to locations of the heap, and Γ is the type context for variables, which is a sequence of typings of the form $x : \tau$ for variables x. The type rules for the basic expression forms are straightforward, and can be found in Appendix A. We focus in the discussion that follows on the expressions aimed at handling interfaces. The model is to be able to create interfaces independently, and eventually to name a particular interface the entry point to a component. An interface is created via the expression $\langle I \mid l_1{=}e_1, \ldots \mid I_1{\leftarrow}e'_1, \ldots \rangle$, where I is the IID of the interface being created, the l_i's are labels of the fields to which the values e_i's are assigned, and the I_i's are the interfaces reachable from the interface being created. The typing rule is straightforward:

$$\frac{\forall i\ \ \mathcal{I}; \Psi; \Gamma \vdash e_i : \tau_i \quad \forall j\ \ \mathcal{I}; \Psi; \Gamma \vdash e'_j : [I_j, \iota_j]}{\mathcal{I}; \Psi; \Gamma \vdash \langle I \mid l_1{=}e_1, \ldots \mid I_1{\leftarrow}e'_1, \ldots \rangle : [I, \langle I_1{:}\iota_1, \ldots \rangle_t]}$$

(under the condition that $\mathcal{I}(I) = \{l_1{:}\tau_1, \ldots\}$, and the tag $t \neq \perp$ is fresh). Note that the tag is assigned nondeterministically by the system.

Given an interface e, you can read a value from a field of e by selection, $e.l$, just like you would a normal record. To follow the link to a connected interface with IID I, the expression $e\#I$ returns the corresponding interface value. The corresponding typing rules are as expected:

$$\frac{\mathcal{I}; \Psi; \Gamma \vdash e : [I, \iota]}{\mathcal{I}; \Psi; \Gamma \vdash e.l : \tau} \text{ if } \mathcal{I}(I) = \{l{:}\tau, \ldots\}$$

$$\frac{\mathcal{I}; \Psi; \Gamma \vdash e : [I', \langle I{:}\iota, \ldots \rangle]}{\mathcal{I}; \Psi; \Gamma \vdash e\#I : [I, \iota]}$$

We already noted that most often, interfaces have a cyclic structure. To construct cycles in interfaces, we use a **rec** construct:

$$\textbf{rec}\ x_1{=}d_1, \ldots\ \textbf{in}\ e$$

with the following interpretation: each declaration d_i represents an interface (i.e., it uses the interface construction form), and it can refer to variables $x_1, \ldots, x_n$. Intuitively, those variables will refer to the other interfaces in the resulting constructed interfaces. We impose the syntactic requirement that all references to variables on the right-hand side of a variable binding in a **rec** construct must occur inside an interface constructor $\langle \mid \mid \rangle$. Consider the cyclic example given earlier:

$$\begin{array}{l} \textbf{rec} \\ \quad x_1 = \langle I_1 \mid l_1 = 10 \mid I_2{\leftarrow}x_2 \rangle \\ \quad x_2 = \langle I_2 \mid l_2 = 20 \mid I_1{\leftarrow}x_1 \rangle \\ \textbf{in} \\ \quad x_1 \end{array}$$

Since the **rec** form constructs potentially cyclic interfaces, the interfaces created get a recursive interface type, as captured by the following typing rule:

$$\frac{\begin{array}{c} \forall j\ \ \mathcal{I}; \Psi; \Gamma, x_i{:}[\mathit{IUnknown}, \alpha_i]^{\,i \in 1..n} \vdash d_j : [I_j, \iota_j] \\ \mathcal{I}; \Psi; \Gamma, x_i{:}[I_i, \mu_i(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n)]^{\,i \in 1..n} \vdash e : \tau \end{array}}{\mathcal{I}; \Psi; \Gamma \vdash \textbf{rec}\ x_1{=}d_1, \ldots, x_n{=}d_n\ \textbf{in}\ e : \tau}$$

Note that the interface selection $e\#I$ only works with interface with an interface type of the form $\langle I_1{:}\iota_1, \ldots \rangle$, and will not work on recursive interface types. Rather than extending the scope of the interface selection operators, we use the observation that a recursive type and its unwinding are equivalent. Instead of allowing this unwinding to happen at arbitrary points during the type derivation, we force an explicit transformation of the recursive type through an **unroll** operator. (In fact, an infinite family, as we index by the rolled-up type.) The rule for **unroll** simply witnesses the type equivalence:

$$\frac{\mathcal{I}; \Psi; \Gamma \vdash e : [I, \mu_i(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n)]}{\mathcal{I}; \Psi; \Gamma \vdash \textbf{unroll}_\iota\ (e) : [I, \iota_i\,[\mu_j(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n)/\alpha_j]]}$$

(when $\iota = \mu_i(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n)$).

[1] Actually, because λ abstraction requires a type, and because types for interfaces contain tags assigned by the system, we have a special tag that means essentially "not specified" ($\perp$). Interfaces appearing in the type of λ abstractions are required to be tagged with $\perp$. An interface type with such a tag will match any similar interface type with some other tag. Again, we will return to this point Section 3.3.

Note that since we never need to roll back an unrolled type, we have no **roll** construct in our language.

3.3 Components and COM conformance

The expressions we described in the previous section allows for a general handling of interfaces. We now focus on the problem of putting interfaces together into components, in the sense of COM. Roughly speaking, legal COM components should be sensible with respect to going from interface to interface. Put it yet another way: a component should implement a given set of interfaces. Now, there may be different implementations *within the same component* of the same interface, and generally, the graph of interface implementation can be arbitrarily large. A legal COM component gives the illusion of having a single implementation of each interface. Moreover, as we saw, querying an interface of the component for *IUnknown* should always give us back the same pointer (the same implementation of the *IUnknown* interface), allowing the *IUnknown* pointer to be used to determine object identity; if two interfaces give you back the same pointer when queried for *IUnknown*, the two interfaces are implemented by the same component.

The idea is simply, given a set of connected interfaces, to isolate a particular interface as being the "entry point" into the component. This interface is required to be an *IUnknown* interface. When this interface is isolated to form the component, moreover, it is statically checked to ensure that is satisfies the requirements for COM conformance. (This check is conservative, in that it will reject components that are legal components, but that cannot be proved so using the typing rules.) The construct **component** (e) takes an *IUnknown* interface e of interface type ι, and returns a *component* of type $[\iota]$. The corresponding typing rule contains judgments that check for COM conformance, that we will describe shortly:

$$\frac{\begin{array}{l}\mathcal{I};\Psi;\Gamma \vdash e : [\mathit{IUnknown}, \iota] \\ \vdash \iota \triangleright (I_1, \ldots, I_k) \\ \vdash \iota \Downarrow t \quad t \neq \bot\end{array}}{\mathcal{I};\Psi;\Gamma \vdash \textbf{component}\ (e) : [\iota]}$$

Of course, since a component is just a particular interface, we have a construct to view a component as that particular interface. Hence, if e is a component, then $e\#$ returns the corresponding *IUnknown* interface, with the obvious typing rule:

$$\frac{\mathcal{I};\Psi;\Gamma \vdash e : [\iota]}{\mathcal{I};\Psi;\Gamma \vdash e\# : [\mathit{IUnknown}, \iota]}$$

Intuitively, typing a component means typing the corresponding interface, which must be a *IUnknown* interface, checking that the interfaces reachable from the *IUnknown* interface form a consistent set (that is, makes all the same interfaces available irrespectively of how one got there), and checking that the *IUnknown* pointers all agree. To achieve this, two new type judgments are introduced. The typing rules corresponding to those judgments are given in Appendix A.

The first judgment, of the form $\Delta \vdash \iota \triangleright (I_1, \ldots, I_k)$, simply checks that all the interfaces reachable from an interface with interface type ι all present interfaces with IID $I_1, \ldots, I_k$. Δ is a list of type variables $\alpha \triangleright (I_1, \ldots, I_m)$ indicating the assumptions on those type variables when type-checking a recursive type. Intuitively, this judgment captures the fact that all the interfaces of a component must present the same interfaces. Note that there is no requirement that these interfaces be the same interfaces; they just need to provide an interface of the appropriate interface type. For example, the following expression meets this requirement:

$$\begin{array}{l}\textbf{rec} \\ \quad x_1 = \langle I_1 \mid l_1 = 10 \mid I_1 \leftarrow x_1, I_2 \leftarrow x_2 \rangle \\ \quad x_2 = \langle I_2 \mid l_2 = 20 \mid I_1 \leftarrow x_1, I_2 \leftarrow x_2 \rangle \\ \textbf{in} \\ \quad x_1\end{array}$$

and so does the following:

$$\begin{array}{l}\textbf{rec} \\ \quad x_1 = \langle I_1 \mid l_1 = 30 \mid I_1 \leftarrow x_1, I_2 \leftarrow x_3 \rangle \\ \quad x_2 = \langle I_2 \mid l_2 = 50 \mid I_1 \leftarrow x_4, I_2 \leftarrow x_2 \rangle \\ \quad x_3 = \langle I_2 \mid l_2 = 40 \mid I_1 \leftarrow x_1, I_2 \leftarrow x_2 \rangle \\ \quad x_4 = \langle I_1 \mid l_1 = 60 \mid I_1 \leftarrow x_1, I_2 \leftarrow x_2 \rangle \\ \textbf{in} \\ \quad x_1\end{array}$$

The second judgment, of the form $\Delta \vdash \iota \Downarrow t$, checks that the given interface type ι is such that any *IUnknown* interface reachable from ι is tagged with the same tag t. Since whenever an interface is created, its type gets a fresh tag, this ensures that all *IUnknown* interfaces reachable are the *same* interface. Part of the difficulty this introduces, as alluded to earlier, is that it doesn't interact well with functional abstraction. One way to explain this is simply as an aliasing problem under a different guise. We solve the problem by a rather crude approach: whenever an interface goes through a functional abstraction as an argument, its tag is cleared (i.e., set to $\bot$).[2] This is not as restrictive as it appears, and most importantly, does *not* prevent us to use first-class interfaces. Since the only place where tags are checked is at the time of component creation, our restriction simply means that when an interface is passed to a function, it cannot be used to construct a component within that function.

4. COMPUTATIONAL SEMANTICS

The operational behavior of λ^{COM} is heavily inspired by the allocation semantics of [5, 6] which makes the allocation of data in the heap explicit. The only pieces of data that we will model as living in the heap are the interfaces. Intuitively, this is because we want to explicitly reason about interface sharing, and identify pointer equality in some cases.

The operational semantics of our calculus (Figure 2) is given by a deterministic rewriting system $M \rightarrow M'$ mapping machine states to new machine states. A machine state consists of a triple (H, e, S) of a heap H, an expression e being executed, and a stack S. A heap is a finite mapping of locations (ℓ) to interfaces, where an interface is simply implemented as a block of memory, containing the data and the location of the other interfaces accessible from that interface. Typing judgments are introduced to assign a type to heaps, as well as machine states. Those judgments are given in Appendix A.

Let us spent some time on the basic infrastructure of the reduction rules. First, consider the management of the stack. Two rules handle this. The first rule pushes a context on the stack:

$$(H, F[e], S) \longrightarrow (H, e, F :: S)$$

Intuitively, if the current expression is a context F whose hole is filled by expression e, then the context F is pushed on the stack while e is evaluated. Eventually, e will evaluate down to an extended value u, at which point the context on top of the stack is

[2]This is achieved in Appendix A using a judgment $\vdash \tau \equiv_{\perp} \tau'$ where τ' is restricted to only have $\bot$ tags.

$$
\begin{array}{lrcl}
\textit{extended values} & u & ::= & i \mid \lambda x{:}\tau.e \mid \ell \mid \textbf{component}\ (\ell) \mid x \\
\\
\textit{heaps} & H & ::= & \{\ell_1 \mapsto h_1, \ldots\} \\
\textit{heap types} & \Psi & ::= & \{\ell_1 : \tau_1, \ldots\} \\
\textit{heap values} & h & ::= & \langle I \mid l_1 = v_1, \ldots \mid I_1{\leftarrow}\ell_1, \ldots\rangle \\
\\
\textit{stack frames} & F & ::= & [\,]\ e \mid v\ [\,] \mid [\,].l \mid [\,]\#I \mid \textbf{rec}\ x_1 = [\,], \ldots\ \textbf{in}\ e \mid \cdots \mid \\
 & & & \textbf{unroll}_\iota\ ([\,]) \mid \langle I \mid l_1 = [\,], \ldots \mid I_1{\leftarrow}e, \ldots\rangle \mid \cdots \mid \\
 & & & \langle I \mid l_1 = u_1, \ldots \mid I_1 \leftarrow [\,], \ldots\rangle \mid \cdots \mid \\
 & & & \textbf{component}\ ([\,]) \mid [\,]\# \\
\textit{stacks} & S & ::= & \text{nil} \mid F :: S \\
\textit{machine states} & M & ::= & (H, e, S)
\end{array}
$$

$$
\begin{array}{rcl}
(H, u, F :: S) & \longrightarrow & (H, F[u], S) \\
(H, F[e], S) & \longrightarrow & (H, e, F :: S) \\
(H, (\lambda x{:}\tau.e)\ u, S) & \longrightarrow & (H, e\,[u/x]\,, S) \\
(H, \langle I \mid l_1 = u_1, \ldots \mid I_1{\leftarrow}u_1', \ldots\rangle, S) & \longrightarrow & (H \oplus \{\ell \mapsto \langle I \mid l_1 = u_1, \ldots \mid I_1{\leftarrow}u_1', \ldots\rangle\}, \ell, S) \quad \text{if } \ell \notin \text{Dom}(H) \\
(H, \ell.l, S) & \longrightarrow & (H, u, S) \quad \text{if } H(\ell) = \langle I \mid l = u, \ldots \mid \ldots\rangle \\
(H, \ell\#I, S) & \longrightarrow & (H, u, S) \quad \text{if } H(\ell) = \langle I \mid \ldots \mid I{\leftarrow}u, \ldots\rangle \\
(H, \textbf{unroll}_\iota\ (u), S) & \longrightarrow & (H, u, S) \\
(H, \textbf{rec}\ x_1 = u_1, \ldots\ \textbf{in}\ e, S) & \longrightarrow & ((H \oplus \{\ell_i \mapsto u_i\})[\ell_1/x_1, \ldots], e[\ell_1/x_1, \ldots], S) \quad \text{if } \ell_1, \ldots \notin \text{Dom}(H) \\
(H, \textbf{component}\ (\ell)\#, S) & \longrightarrow & (H, \ell, S)
\end{array}
$$

Figure 2: λ^{COM} operational semantics

popped, and filled with the value u:

$$(H, u, F :: S) \longrightarrow (H, F[u], S).$$

Notice that the stack is popped when an expression is reduced to an *extended value*, not simply a value. Essentially, an extended value is either a value or an unevaluated variable. Such unevaluated variables occur during the interpretation of **rec** constructs: the bindings are evaluated without resolving the variables. When the **rec** bindings are all evaluated, the variables are then resolved.

The terminal states of the reduction relation are states of the form (H, v, nil), that is, yielding a value v, with an empty stack indicating that nothing remains to be done. The type system ensures that the evaluation of a well-typed expression never enters a *stuck state*. A state (H, e, S) is *stuck* if e is not a value and there does not exist (H', e', S') such that $(H, e, S) \rightarrow (H', e', S')$. The following theorem follows from Preservation and Progress lemmas, *à la* Wright and Felleisen [18]:

THEOREM 4.1 (TYPE SOUNDNESS). *If $\mathcal{I} \vdash (H, e, nil) : \tau$ and $(H, e, nil) \longrightarrow^* (H', e', S')$ then (H', e', S') is not stuck.*

Type soundness is one property that we want from a type system in general. Our type system, as we saw above, also seeks to ensure that created components are conformant to the rules of COM. In order to address this issue, we need to formalize some of the concepts we need. Given a heap H, and given an I interface pointer ℓ (that is, a pointer ℓ such that $H(\ell) = \langle I \mid \cdots \mid \cdots\rangle$), we say that an I' interface pointer ℓ' is *immediately reachable* from ℓ (written $\ell \rightsquigarrow_H \ell'$) if $H(\ell) = \langle I \mid \cdots \mid I'{\leftarrow}\ell', \ldots\rangle$. We say ℓ' is *reachable* from ℓ if there exists a sequence $\ell_1, \ldots, \ell_n$ such that $\ell \rightsquigarrow_H \ell_1 \rightsquigarrow_H \cdots \rightsquigarrow_H \ell_n \rightsquigarrow_H \ell'$. Corresponding to a component C, we have its *IUnknown* interface pointer ℓ^C. We say an interface pointer ℓ is *an interface of* C if $\ell^C \rightsquigarrow_H^* \ell$. We can now formalize the properties implying COM conformance.

An *IUnknown* interface pointer ℓ^C (representing a component C) is *COM conformant* (with respect to a heap H) if the following properties hold:

- if ℓ if an interface of C, then ℓ^C is immediately reachable from ℓ,
- if ℓ_1 is an I interface of C, then there exists an I interface pointer ℓ_2 immediately reachable from ℓ_1,
- if ℓ_1 is an I_1 interface of C and ℓ_2 is an I_2 interface of C immediately reachable from ℓ_1, then there exists an I_1 interface ℓ_3 immediately reachable from ℓ_2, and
- if ℓ_1 is an I_1 interface of C, ℓ_2 is an I_2 interface of C immediately reachable from ℓ_1, and ℓ_3 is an I_3 interface of C immediately reachable from ℓ_2, then there exists an I_3 interface ℓ_4 immediately reachable from ℓ_1.

These rules capture the required properties described in Section 2. Also, the above makes clear that not all the interfaces need be implemented by the same pointers. We can now prove that our type system indeed enforces COM conformance of components.

THEOREM 4.2. *If $\mathcal{I} \vdash (H, \textbf{component}\ (\ell), S) : \tau$, then ℓ is COM conformant with respect to H.*

Intuitively, the typing rules guarantee that if $(H, \textbf{component}\ (\ell), S)$ type-checks, then **component** (ℓ) itself type-checks, and the typing rule for **component** (ℓ) involves judgments to ensure that ℓ has the required properties. A variation on type soundness leads to the following corollary of Theorem 4.2, showing that it is essentially sufficient for COM conformance that **component** (e) type-checks:

COROLLARY 4.3. *If* $\mathcal{I} \vdash (H, \textbf{\textit{component}}\ (e), nil) : \tau$ *and* $(H, \textbf{\textit{component}}\ (e), nil) \longrightarrow^* (H', \textbf{\textit{component}}\ (\ell), nil)$*, then* ℓ *is COM conformant with respect to* H'*.*

5. EXTENSIONS

Several natural extensions to the basic framework are possible, and deserve to be explored further. We outline the main ones here, relegating their development to future work.

5.1 Subtyping of interfaces

One natural extension of λ^{COM} is to allow subtyping. Although we can extend λ^{COM} itself to a fully subtyped calculus, we can restrict our attention to subtyping of interfaces, more precisely subtyping of the reachable interfaces of an interface. In other words, we would like to say, for instance, that an interface of type $[I, \langle I_1{:}\iota_1\rangle]$ is a subtype of $[I, \langle I_1{:}\iota_1, I_2{:}\iota_2\rangle]$. Why only look at subtyping in terms of reachable interfaces? Presumably, depth and width subtyping of the fields of an interface could be achieved by moving to a full subtyping calculus. However, the interfaces of COM are immutable, and so not allowing subtyping on fields seems more in spirit with the model. The aim is to be able to write functions that expect an interface I from which only such and such interfaces can be reached—for instance, only those interfaces used within the body of the functions. The type of the interface argument of these functions then captures the minimal requirements of the interfaces

To extend λ^{COM} in that way, we need to define a subtyping relation over interface types. This turns out to be tricky because of the presence of type of the form $\mu_i(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n)$, which compute multiple fixed points for recursive types. As a first approximation, and to capture the case of interest above, we can restrict our attention to the case where the bound type is not recursive. Thus, we could define a new type judgment $\vdash \iota_1 \leq \langle I_1 : \iota_1, \ldots\rangle$:

$$\frac{\vdash \iota_{I_i} \leq \iota'_{I_i}}{\vdash \langle I_1 : \iota_{I_1}, \ldots\rangle \leq \langle I'_1 : \iota'_{I'_1}, \ldots\rangle} \text{ if } \{I_1, \ldots\} \subseteq \{I'_1, \ldots\}$$

$$\frac{\vdash \iota_i[\mu_1(\alpha_1 \ldots \alpha_n).(\iota_1, \ldots, \iota_n)/\alpha_1, \ldots] \leq \iota}{\vdash \mu_i(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n) \leq \iota}$$

To simplify our life, we require an explicit coercion when a supertype is needed. (The coercion would not change the runtime representation of the interface, which is after all simply a location in the heap, only the type.) The functions $\mathbf{up}_\iota\ (e)$, indexed by the target type, perform the necessary translation:

$$\frac{\mathcal{I}; \Psi; \Gamma \vdash e : [I, \iota_1] \quad \vdash \iota_1 \leq \iota_2}{\mathcal{I}; \Psi; \Gamma \vdash \mathbf{up}_{\iota_2}\ (e) : [I, \iota_2]}$$

It should be clear that such an extension to λ^{COM} is still sound. COM conformance still holds, although under a weakened form; the type system isolates a conformant subset of interfaces. So the component itself may not be strictly speaking conformant, but what can effectively be reached according to the type system is conformant.

5.2 Dynamic interface negotiation

The most noticeable discrepancy between λ^{COM} and the general COM model is that we require complete knowledge of the components under consideration. Thus, we can handle any component we create, as the construction will force the type to reveal the available interfaces. In contrast, COM allows the client to use a COM component it knows nothing about, by querying it for interfaces. Our type system only allows a client to query for a component which is statically known to provide the interface. COM allows for dynamic queries: querying for an interface returns an interface only if such an interface is available; if it is not, no failure occurs, but an indication is returned to the user. This allows processing to be tailored to the component at hand, a special case of this being "graceful fallback" behavior: if an old interface I_1 is updated into a new better and faster I_2, an application can try to see if a component implementing I_1 implements I_2 (under the assumption that it would use that interface if available), and fall back on I_1 if not.

It is easy to extend our calculus to account for some kind of imported component for which nothing is known initially. We thus need a construct to dynamically query for an interface. The construct **case** $I \in x\ (e_1 \mid e_2)$ tests if the interface accessed through the variable x supports interface I; if so, it evaluates e_1, otherwise it evaluates e_2. This check is performed in some unspecified way for imported components. This construct can also be used (with the appropriate operational behavior) as a *downcasting* operator in conjunction with subtyping (cf. Section 5.1): if an application of $\mathbf{up}_\iota\ (e)$ has "forgotten" some interfaces existing in e, it is possible to query back for them. We restrict the checking to a variable for technical reasons that will become clear after we look at the typing rule.

$$\frac{\mathcal{I}; \Psi; \Gamma \oplus x{:}[I_0, \langle I_1{:}\iota_1, \ldots, I_n{:}\iota_n, I{:}\langle\rangle\rangle] \vdash e_1 : \tau' \quad \mathcal{I}; \Psi; \Gamma \vdash e_2 : \tau'}{\mathcal{I}; \Psi; \Gamma \vdash \textbf{case}\ I \in x\ (e_1 \mid e_2) : \tau'}$$

(if $\Gamma(x) = [I_0, \langle I_1{:}\iota_1, \ldots, I_n{:}\iota_n\rangle]$, and where $\oplus$ indicates environment update). The idea is simply that to type e_1, we can assume that the type of x has been updated to reflect the fact that interface I is indeed available. This explains the restriction to variables; we need a place where we can attach the new information.

Again, the resulting calculus is easily seen to be sound. What about COM conformance? Unfortunately, we lose the ability to check for conformance of components that use dynamic negotiation. A problem, for instance, is to ensure that components are stable in the sense of Section 2. One approach would be to define a version of conformance that is dependent on the conformance of imported components; this venue however remains to be explored.

5.3 Aggregation

Up to a point, we can encompass in our model both subtyping of interfaces and dynamic interface negotiation. A harder aspect of COM to model is aggregation, one of the ways of composing components we saw in Section 2. (Containment, the other composition method, is in some sense trivial to implement, as it is a question of locally allocating a component within the scope of another component.) There are two main problems with modeling aggregation. First, it is not immediately clear how to account for aggregation at the time of component creation. The COM framework requires a component to know when it is being created in aggregated form. This permits the aggregated component to implement a special version of its *QueryInterface* operation, to account for the rules of aggregation (see [4] for a description of those rules). The implementation of aggregation is heavily biased towards a single implementation of *QueryInterface*, shared across all the interfaces of a component. In our model, where interfaces are created in a sense separately from components, aggregation requires modifying already allocated interfaces at aggregation time. Accommodating such modifications in our framework seems to

require moving to a setting where interface allocation and initialization is dissociated, in the style of [12, 17]. We are currently exploring this approach.

Even if the basic model can be modified to handle aggregation, the problem of deciding COM conformance still remains. In the presence of dynamic negotiation, one of the results of [14] is that COM conformance in the presence of aggregation cannot be captured statically. Roughly speaking, a component is COM conformant with respect to aggregation if it is correct with respect to all "valid" sequences of operations on the component. (The definition of a valid sequence of operation can be found in [14].) This seems to force the check for COM conformance to be performed at runtime.

6. RELATED WORK

The most relevant work related to our work, in that it attempts to capture the COM framework at a linguistic level and tries to prove properties of the framework in that setting, is that of Ibrahim and Szyperski [2, 3]. They define a language (the COM External Language, or COMEL), a Modula-like language with primitive notions of interfaces, containment and aggregation. They decide to work at a higher level than we do, abstracting away many of the details of COM; for instance, they completely subsume the *QueryInterface* mechanism at the language level. As a consequence, they cannot reason about aspects of COM that rely on an explicit pointer representation, such as the COM conformance of a set of interfaces.

We have already mentioned the work of Sullivan *et al.* [14, 15]. Their approach to formalizing COM is totally different. They express the properties of the framework in a formal language, the Z notation [13], and derive properties and requirements of COM conformant components. They do not attempt to provide an execution model for the framework.

Other approaches to formalization are not specific to COM, but attempt to get at the essence of components. Many such approaches are derived straight from object-oriented developments. For instance, Seco and Caires [11] describe a calculus that captures what are, in their view and others, the basic elements of component-based programming, namely explicit context dependency, dynamic binding, subtype polymorphism, dynamic composition, and object composition. Although some of those issues arise in COM, the calculus as presented is at a much higher level of abstraction than λ^{COM}. Moreover, λ^{COM} does not at the present time attempt to get at the generalities of component-oriented programming, but rather aims at capturing the key elements of the COM model, a different goal. It is hoped that eventually, higher level issues can be derived from the low level description of λ^{COM}.

An interesting direction in recent work on component-oriented programming is the development of type systems that capture the notion of "contract" with a component [16]. Intuitively, using a component correctly in a given context requires the context to follow a given protocol to interact with the component: maybe functions need to be called in some order, such as *open* before *read*, etc. Composing components requires reasoning about the interaction of those contracts. Work by Reussner and others [8, 9] attempts to develop type systems in those directions. As we noted before, this work is at a much higher level than ours, and in fact is not incompatible with our approach.

7. DISCUSSION

We have described in this paper a first step in the direction of a general calculus for reasoning about COM-style components and interfaces. Such a calculus is a requirement for deriving programming languages based on the COM model, with the hopes of a type system to statically guarantee conformance of the created components with the requirements of the component framework. (A tentative step in that particular direction is outlined in [7]; attempting to get the framework in that paper to work highlighted the need for the formal work described in this paper.)

Our approach makes a number of simplifying assumptions. The central design issue of our calculus, in fact, of any formal framework intending to model COM, is the design of the *QueryInterface* functionality. Recall that in COM, *QueryInterface* is implemented by a method present in every interface. In our calculus, the functionality is implemented by what amounts to a lookup table—clearly a simplification. Among other things, this means that given an interface pointer ℓ, querying ℓ for an interface I will always return the same pointer. In contrast, in COM, the interface pointer returned can be different. (For the sake of an example, consider a *QueryInterface* implementation that when queried for a particular interface alternatively returns one of two interfaces pointers, maybe to reduce the load on whichever machine implements the actual interface in a distributed setting.) Presumably, our calculus can be modified to handle this, but this would make checking for conformance more difficult. One possibility would be to allow a *QueryInterface* function in each interface, expressed from within λ^{COM}, expecting an interface name and returning an interface pointer. To check for COM conformance, we then need to be able to statically specify the interface names for which the function returns an interface pointer.

Another restriction, this time syntactic, concerns the handling of recursive interfaces. Recursive interfaces must be defined within a same **rec** block. Moreover, the fields in the interface cannot depend on the interfaces being recursively defined. This restriction can be somewhat weakened, but for the purposes of this paper, we can use the semantics and rules we set forth. It is also possible to move to an even lower level operational semantics, that dissociates the allocation of space in the heap for interfaces from the initialization of its fields and reachable interfaces. This approach would lead to a calculus in the style of [12, 17], and seems required to deal with aggregation.

One restriction that is *not* imposed by our system is the handling of *QueryInterface* in a centralized way. In most COM implementations, typically based on object-oriented languages such as C++, interfaces are objects, and inherit from a base object *IUnknown*. The *IUnknown* object implements the *QueryInterface* function. This ensures COM conformance, as every interface automatically recognizes the same interfaces as every other. We chose not to model things that way for the sake of generality. Besides, even in the C++ implementation, an object inheriting from *IUnknown* is allowed to redefine *QueryInterface* and therefore throws one back to the full generality of the COM model.

In our calculus, we chose not to model components in any special way. In fact, what we are calling components in Section 3.3 are more accurately called component instances. Future work clearly points to handling components as makers or constructors (maybe via actual class factories). Again, this may be required to deal with aggregation correctly.

There remains much work to be done to cover even the basic COM model. The most important issues include modeling conformance in the presence of dynamic interface negotiation (cf. Section 5.2), and modeling aggregation (cf. Section 5.3). Finally, another central aspect of the COM model not addressed in existing formal accounts is the issue of memory management: COM gives very explicit rules for allocating and deallocating component instances. In fact, the *IUnknown* interface prescribes two other

methods aside from *QueryInterface*, namely *AddRef* and *Release* to perform reference-counting memory management for component instances. It would be interesting to model this in our calculus via explicit memory management [5], and attempt to formally prove the appropriateness of the rules for COM memory management.

8. ACKNOWLEDGEMENTS

I have greatly benefitted from discussions with Greg Morrisett. Dan Grossman and David Walker have read early drafts of this work and provided helpful comments.

9. REFERENCES

[1] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[2] R. Ibrahim and C. Szyperski. The COMEL language. Technical Report FIT-TR-97-06, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, 1997.

[3] R. Ibrahim and C. Szyperski. Can Component Object Model (COM) be formalized? — the formal semantics of the COMEL language. Work-In-Progress, IRW/FMP'98. Also appears as Technical Report TR-CS-98-09, The Australian National University, 1998.

[4] Microsoft Corporation and Digital Equipment Corporation. The Component Object Model Specification. Draft version 0.9, available from `http://www.microsoft.com/com`, 1995.

[5] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *ACM Conference on Functional Programming and Computer Architecture*, pages 66–77. ACM Press, 1995.

[6] G. Morrisett and R. Harper. Semantics of memory management for polymorphic languages. In *Higher-Order Operational Techniques in Semantics*, pages 175–226. Cambridge University Press, 1997.

[7] R. Pucella. The design of a COM-oriented module system. In *Proceedings of the Joint Modular Languages Conference*, number 1897 in Lecture Notes in Computer Science, pages 104–118. Springer-Verlag, 2000.

[8] R. Reussner. Dynamic types for software components. In *Companion of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)*, 1999.

[9] R. Reussner and D. Heuzeroth. A Meta-Protocol and Type system for the Dynamic Coupling of Binary Components. In *Proceedings of the OOPSLA'99 Workshop on Object Oriented Reflection and Software Engineering*, 1999.

[10] D. Rogerson. *Inside COM*. Microsoft Press, 1997.

[11] J. C. Seco and L. Caires. A basic model of typed components. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1850 of *Lecture Notes in Computer Science*, pages 108–128, 2000.

[12] F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proceedings of the European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, 2000.

[13] M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science. Prentice-Hall, second edition, 1992.

[14] K. J. Sullivan and M. Marchukov. Interface negotiation and efficient reuse: A relaxed theory of the component object model. Technical Report 97-11, Department of Computer Science, University of Virginia, 1997.

[15] K. J. Sullivan, M. Marchukov, and J. Socha. Analysis of a conflict between aggregation and interface negotiation in Microsoft's Component Object Model. *IEEE Transactions on Software Engineering*, 25(4):584–599, 1999.

[16] C. Szyperski. *Component Software*. Addison Wesley, 1997.

[17] D. Walker and G. Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, 2000.

[18] A. K. Wright and M . Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

APPENDIX

A. TYPING RULES

We assume a canonical ordering on the fields of an interface, and a compatible canonical ordering on both the list of interfaces reachable from an interface, and the list of interfaces in an interface type.

We assume that the set IIDS of interface identifiers contains the special interface identifier *IUnknown*, and that any interface assignment $\mathcal{I}$ is such that $\mathcal{I}(\mathit{IUnknown}) = \{\}$.

The typing rules also need the following auxiliary functions. The function tag returns the tag of a particular interface:

$$tag(\langle I_1{:}\iota_1, \ldots\rangle_t) = t$$
$$tag(\mu_i(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n)) = tag(\iota_i)$$

The function FTV returns all the free type variables of a given interface type.

We assume that type equivalence is simply syntactic equality up to renaming of bound variables.

$\boxed{\mathcal{I}; \Psi; \Gamma \vdash e : \tau}$

$$\frac{}{\mathcal{I}; \Psi; \Gamma, x{:}\tau \vdash x : \tau} \qquad \frac{}{\mathcal{I}; \Psi; \Gamma \vdash i : \text{int}} \qquad \frac{}{\mathcal{I}; \Psi; \Gamma \vdash \ell : \tau} \text{ if } \Psi(\ell) = \tau$$

$$\frac{\mathcal{I}; \Psi; \Gamma, x{:}\tau \vdash e : \tau'}{\mathcal{I}; \Psi; \Gamma \vdash \lambda x{:}\tau.e : \tau \to \tau'} \qquad \frac{\mathcal{I}; \Psi; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \mathcal{I}; \Psi; \Gamma \vdash e_2 : \tau_3 \quad \vdash \tau_3 \equiv_\perp \tau_1}{\mathcal{I}; \Psi; \Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\forall j\ \ \mathcal{I}; \Psi; \Gamma, x_i{:}[\mathit{IUnknown}, \alpha_i]^{\,i\in 1..n} \vdash d_j : [I_j, \iota_j] \quad \mathcal{I}; \Psi; \Gamma, x_i{:}[I_i, \mu_i(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n)]^{\,i\in 1..n} \vdash e : \tau}{\mathcal{I}; \Psi; \Gamma \vdash \textbf{rec}\ x_1 = d_1, \ldots, x_n = d_n\ \textbf{in}\ e : \tau}$$

$$\frac{\forall i\ \ \mathcal{I}; \Psi; \Gamma \vdash e_i : \tau_i \quad \forall j\ \ \mathcal{I}; \Psi; \Gamma \vdash e'_j : [I_j, \iota_j]}{\mathcal{I}; \Psi; \Gamma \vdash \langle I \mid l_1 = e_1, \ldots \mid I_1{\leftarrow}e'_1, \ldots\rangle : [I, \langle I_1 : \iota_1, \ldots\rangle_t]} \text{ if } \mathcal{I}(I) = \{l_1{:}\tau_1, \ldots\} \text{ and } t \neq \perp \text{ is fresh}$$

$$\frac{\mathcal{I}; \Psi; \Gamma \vdash e : [I, \iota]}{\mathcal{I}; \Psi; \Gamma \vdash e.l : \tau} \text{ if } \mathcal{I}(I) = \{l{:}\tau, \ldots\} \qquad \frac{\mathcal{I}; \Psi; \Gamma \vdash e : [I', \langle I : \iota, \ldots\rangle]}{\mathcal{I}; \Psi; \Gamma \vdash e\#I : [I, \iota]}$$

$$\frac{\mathcal{I}; \Psi; \Gamma \vdash e : [I, \mu_i(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n)]}{\mathcal{I}; \Psi; \Gamma \vdash \textbf{unroll}_\iota\ (e) : [I, \iota_i\,[\mu_j(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n)/\alpha_j]]} \text{ if } \iota = \mu_i(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n)$$

$$\frac{\mathcal{I}; \Psi; \Gamma \vdash e : [\mathit{IUnknown}, \iota] \quad \vdash \iota \triangleright (I_1, \ldots, I_k) \quad \vdash \iota \Downarrow t \quad t \neq \perp}{\mathcal{I}; \Psi; \Gamma \vdash \textbf{component}\ (e) : [\iota]}$$

$$\frac{\mathcal{I}; \Psi; \Gamma \vdash e : [\iota]}{\mathcal{I}; \Psi; \Gamma \vdash e\# : [\mathit{IUnknown}, \iota]}$$

$\boxed{\vdash \tau \equiv_\perp \tau'}$

$$\frac{}{\vdash \text{int} \equiv_\perp \text{int}} \qquad \frac{}{\vdash \tau_1 \to \tau_2 \equiv_\perp \tau_1 \to \tau_2} \qquad \frac{\vdash \iota_1 \equiv_\perp \iota_2}{\vdash [I, \iota_1] \equiv_\perp [I, \iota_2]}$$

$\boxed{\vdash \iota \equiv_\perp \iota'}$

$$\frac{\vdash \iota_1 \equiv_\perp \iota'_1 \quad \ldots}{\vdash \langle I_1{:}\iota_1, \ldots\rangle_t \equiv_\perp \langle I_1{:}\iota'_1, \ldots\rangle_\perp} \qquad \frac{}{\vdash \alpha \equiv_\perp \alpha}$$

$$\frac{\vdash \iota_1 \equiv_\perp \iota'_1 \quad \ldots \quad \vdash \iota_n \equiv_\perp \iota'_n}{\vdash \mu_i(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n) \equiv_\perp \mu_i(\alpha_1, \ldots, \alpha_n).(\iota'_1, \ldots, \iota'_n)}$$

$\boxed{\mathcal{I}; \Psi; \Gamma \vdash F : \tau_1 \to \tau_2; \Gamma'}$

$$\frac{\mathcal{I}; \Psi; \Gamma \vdash e : \tau_1}{\mathcal{I}; \Psi; \Gamma \vdash [\,]\ e : (\tau_1 \to \tau) \to \tau; \Gamma} \qquad \frac{\mathcal{I}; \Psi; \Gamma \vdash v : \tau_1 \to \tau_2}{\mathcal{I}; \Psi; \Gamma \vdash v\ [\,] : \tau_1 \to \tau_2; \Gamma}$$

$$\frac{}{\mathcal{I}; \Psi; \Gamma \vdash [\,].l : [I, \iota] \to \tau; \Gamma} \text{ if } \mathcal{I}(I) = \{l : \tau, \ldots\}$$

$$\frac{}{\mathcal{I};\Psi;\Gamma \vdash [\,]\#I : [I', \langle I{:}\iota, \ldots\rangle] \to [I, \iota];\Gamma}$$

$$\frac{}{\mathcal{I};\Psi;\Gamma \vdash \textbf{unroll}_\iota\ ([\,]) : [I, \iota] \to [I, \iota_i[\mu_1(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n)/\alpha_1, \ldots]];\Gamma} \text{ if } \iota = \mu_i(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n)$$

$$\frac{\forall i \neq 1\ \ \mathcal{I};\Psi;\Gamma \vdash e_i : \tau_i \quad \forall j\ \ \mathcal{I};\Psi;\Gamma \vdash e'_j : [I_j, \iota_j]}{\mathcal{I};\Psi;\Gamma \vdash \langle I \mid l_1 = [\,], l_2 = e_2, \ldots \mid I_1{\leftarrow}e'_1, \ldots\rangle : \tau_1 \to [I, \langle I_1{:}\iota_1, \ldots\rangle];\Gamma} \text{ if } \mathcal{I}(I) = \{l_1{:}\tau_1, \ldots\}$$

$$\vdots$$

$$\frac{\forall i\ \ \mathcal{I};\Psi;\Gamma \vdash v_i : \tau_i \quad \forall j \neq 1\ \ \mathcal{I};\Psi;\Gamma \vdash e'_j : [I_j, \iota_j]}{\mathcal{I};\Psi;\Gamma \vdash \langle I \mid l_1 = v_1, \ldots \mid I_1{\leftarrow}[\,], I_2{\leftarrow}e'_2, \ldots\rangle : [I, \iota_1] \to [I, \langle I_1{:}\iota_1, \ldots\rangle];\Gamma} \text{ if } \mathcal{I}(I) = \{l_1{:}\tau_1, \ldots\}$$

$$\vdots$$

$$\frac{\forall j \neq 1\ \ \mathcal{I};\Psi;\Gamma, x_i{:}[\mathit{IUnknown}, \alpha_i]^{\,i\in 1..n} \vdash d_j : [I_j, \iota_j] \quad \mathcal{I};\Psi;\Gamma, x_i{:}[I_i, \mu_i(\alpha_1, \ldots, \alpha_n).(\tau_1, \ldots, \tau_n)]^{\,i\in 1..n} \vdash e : \tau}{\mathcal{I};\Psi;\Gamma \vdash \textbf{rec}\ x_1 = [\,], x_2 = d_2, \ldots\ \textbf{in}\ e : [I_1, \mu_1(\alpha_1, \ldots, \alpha_n).(\tau_1, \ldots, \tau_n)] \to \tau;\Gamma, x_i : [\mathit{IUnknown}, \alpha_i]}$$

$$\vdots$$

$$\frac{}{\mathcal{I};\Psi;\Gamma \vdash \textbf{component}\ ([\,]) : [\mathit{IUnknown}, \iota] \to [\iota];\Gamma}$$

$$\frac{}{\mathcal{I};\Psi;\Gamma \vdash [\,]\# : [\iota] \to [\mathit{IUnknown}, \iota];\Gamma}$$

$\boxed{\mathcal{I};\Psi;\Gamma \vdash S : \tau_1 \to \tau_2;\Gamma'}$

$$\frac{}{\mathcal{I};\Psi;\Gamma \vdash \text{nil} : \tau \to \tau;\Gamma}$$

$$\frac{\mathcal{I};\Psi;\Gamma \vdash S : \tau_2 \to \tau_3;\Gamma' \quad \mathcal{I};\Psi;\Gamma' \vdash F : \tau_1 \to \tau_2;\Gamma''}{\mathcal{I};\Psi;\Gamma \vdash F :: S : \tau_1 \to \tau_3;\Gamma''}$$

$\boxed{\mathcal{I};\Psi;\Gamma \vdash (e, S) : \tau}$

$$\frac{\mathcal{I};\Psi;\Gamma \vdash S : \tau_1 \to \tau_2;\Gamma' \quad \mathcal{I};\Psi;\Gamma' \vdash e : \tau_1}{\mathcal{I};\Psi;\Gamma \vdash (e, S) : \tau_2}$$

$\boxed{\mathcal{I} \vdash H : \Psi}$

$$\frac{\mathcal{I};\Psi \vdash h_i : \tau_i}{\mathcal{I} \vdash \{\ell_1 \mapsto h_1, \ldots\} : \Psi} \text{ if } \Psi = \{\ell_1 : \tau_1, \ldots\}$$

$\boxed{\mathcal{I};\Gamma \vdash (H, e, S) : \tau}$

$$\frac{\mathcal{I} \vdash H : \Psi \quad \mathcal{I};\Psi;\Gamma \vdash (e, S) : \tau}{\mathcal{I};\Gamma \vdash (H, e, S) : \tau}$$

$\boxed{\Delta \vdash \iota \triangleright (I_1, \ldots, I_k)}$

$$\frac{\Delta \vdash \iota \triangleright (I_{p(1)}, \ldots, I_{p(k)})}{\Delta \vdash \iota \triangleright (I_1, \ldots, I_k)} \text{ if } p \text{ is a permutation of } \{1, \ldots, k\}$$

$$\frac{}{\Delta, \alpha \triangleright (I_1, \ldots, I_k) \vdash \alpha \triangleright (I_1, \ldots, I_k)}$$

$$\frac{\Delta \vdash \iota_i \triangleright (I_1, \ldots, I_k)}{\Delta \vdash \langle I_1 : \iota_1, \ldots \rangle \triangleright (I_1, \ldots, I_k)}$$

$$\frac{\Delta, \alpha_{i_1} \triangleright (I_1, \ldots, I_k), \ldots, \alpha_{i_m} \triangleright (I_1, \ldots, I_k) \vdash \iota_i \triangleright (I_1, \ldots, I_k)}{\Delta \vdash \mu_i(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n) \triangleright (I_1, \ldots, I_k)} \text{ if } FTV(\iota_i) = \{\alpha_{i_1}, \ldots, \alpha_{i_m}\}$$

$\boxed{\vdash \iota \downarrow t}$

$$\frac{}{\vdash \langle \mathit{IUnknown} : \iota, \ldots \rangle \downarrow t} \text{ if } tag(\iota) = t$$

$$\frac{\vdash \iota_i[\mu_1(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n)/\alpha_1, \ldots] \downarrow t}{\vdash \mu_i(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n) \downarrow t}$$

$\boxed{\Delta \vdash \iota \Downarrow t}$

$$\frac{}{\Delta, \alpha \Downarrow t \vdash \alpha \Downarrow t}$$

$$\frac{\vdash \langle I_1{:}\iota_1, \ldots \rangle \downarrow t \quad \forall i \;\; \Delta \vdash \iota_i \Downarrow t}{\Delta \vdash \langle I_1{:}\iota_1, \ldots \rangle \Downarrow t}$$

$$\frac{\vdash \mu_i(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n) \downarrow t \quad \forall i \;\; \Delta, \alpha_{i_1} \Downarrow t, \ldots, \alpha_{i_m} \Downarrow t \vdash \iota_i \Downarrow t}{\Delta \vdash \mu_i(\alpha_1, \ldots, \alpha_n).(\iota_1, \ldots, \iota_n) \Downarrow t} \text{ if } FTV(\iota_i) = \{\alpha_{i_1}, \ldots, \alpha_{i_m}\}$$