Gradual Typing for First-Class Classes [*]

Asumu Takikawa T. Stephen Strickland Christos Dimoulas
Sam Tobin-Hochstadt Matthias Felleisen
PLT, Northeastern University
{asumu, sstrickl, chrdimo, samth, matthias}@ccs.neu.edu

Abstract

Dynamic type-checking and object-oriented programming often go hand-in-hand; scripting languages such as Python, Ruby, and JavaScript all embrace object-oriented (OO) programming. When scripts written in such languages grow and evolve into large programs, the lack of a static type discipline reduces maintainability. A programmer may thus wish to migrate parts of such scripts to a sister language with a static type system. Unfortunately, existing type systems neither support the flexible OO composition mechanisms found in scripting languages nor accommodate sound interoperation with untyped code.

In this paper, we present the design of a gradual typing system that supports sound interaction between statically- and dynamically-typed units of class-based code. The type system uses row polymorphism for classes and thus supports mixin-based OO composition. To protect migration of mixins from typed to untyped components, the system employs a novel form of contracts that partially seal classes. The design comes with a theorem that guarantees the soundness of the type system even in the presence of untyped components.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects

General Terms Languages, Design

Keywords gradual typing, first-class classes, contracts, sealing, design by contract, row polymorphism, blame theorem (proof technique)

[*] Supported in part by two NSF grants, the DARPA CRASH program, and a grant from the Mozilla Foundation.

1. Untyped Object-Oriented Style

The popularity of untyped programming languages such as Python, Ruby, or JavaScript has stimulated work on combining static and dynamic type-checking. The idea is now popularly called *gradual typing* [27]. At this point, gradual typing is available for functional programming languages such as Racket [33, 34], for object-oriented languages such as Ruby [12] or Thorn [38], and for Visual Basic [23] on the .NET platform. Proposals for gradual typing also exist for JavaScript [19] and Perl [31]. Formal models have validated soundness for gradual type systems, allowing seamless interoperation between sister languages [22, 27, 32].

```
(define drracket-frame%
 (size-pref-mixin
  (searchable-text-mixin
   (searchable-mixin
    (status-line-mixin
     (text-mixin
      (editor-mixin
       (standard-menus-mixin
        frame%))))))))
```

Figure 1. Abbreviated code with a chain of mixins

Unfortunately, no existing gradual type system supports the full range of object-oriented styles found in scripting languages. These untyped languages tend to support flexible mechanisms for class composition, such as mixins or traits, that allow the programmer to abstract over inheritance. Furthermore, some untyped languages support a generalization of mixins and traits where classes are first-class values and thus can inherit from other classes at runtime. For example, the implementation of the DrRacket IDE [8] makes extensive use of layered combinations of mixins to implement text editing features, as seen in the abbreviated example given in figure 1—the full code uses 17 mixins.

In such languages, class composition requires the programmer to reason about the specialization interfaces [20] of superclasses. A faithful type system must enable the programmer to express this reasoning via types. Meanwhile, a gradually typed language should support the exchange of "editor.rkt"

```
import widget% from "gui.rkt"
(define editor%
  (class widget%
    (super-new)
    (define/public (on-key key)
      (case key
        [("C-x") (do-cut)]
        ...))
    (define/public (on-paint) ...)
    (define/public (save-file name) ...)
    (define/private (do-cut) ...)
    ...))
```

```
(define (modal-mixin base%)
  (class base%
    (super-new)
    (field [mode 'command])
    (define/public (toggle-mode)
      (if (eq? mode 'command)
          (set! mode 'insert)
          (set! mode 'command)))
    (define/override (on-key key)
      (if (eq? mode 'command)
          (on-key/cmd key)
          (on-key/ins key))) ...))
```

Figure 2. Editor module

classes between typed and untyped components while enforcing the type system's invariants. This desideratum implies protecting cross-component inheritance.

To address these issues, we propose a novel gradual type system that properly validates a programmer's reasoning in the presence of both dynamic class composition and inheritance across untyped-typed boundaries. Our design rests on two technical results. First, we introduce *partial run-time sealing* as a mechanism to protect inheritance-related invariants at the boundary between typed and untyped components. Second, we utilize *row polymorphism* to express constraints on class composition at component boundaries. We present examples motivating the design in section 2. Our design comes with a formal model, which we outline in section 3. We address soundness for the system, including typed-untyped interaction, in section 4. [1]

2. Typing & Protecting First-Class Classes

We use Racket—a language with first-class classes [11]—to present our design. Our choice of language is motivated by our practical experience with first-class classes in Racket and the availability of Typed Racket [33], a gradually typed sister language of Racket. While our design is developed with Racket in mind, many of the lessons learned should apply to other languages with dynamic class and object composition, such as Ruby, Python, or JavaScript.

In Typed Racket, untyped and typed code interoperate at the granularity of modules. Interactions between modules are mediated by higher-order behavioral contracts [9]. That is, when exporting values to untyped modules from typed modules, the language turns types into appropriate contracts. In a typed module, the programmer annotates imports from untyped modules with types, that are enforced at runtime via contracts.

[1] For the complete definitions and proofs, see the technical report version of this paper [30].

Unfortunately, Typed Racket does not currently support first-class classes and the programming patterns they support, particularly mixins and traits. While these programming patterns are widely used, they greatly complicate gradual typing. To illustrate the problems, we present a series of examples using (mostly) Racket syntax. The scientific reader will easily recognize similar problems in related languages.

2.1 Programming with First-Class Classes

The architectures of GUI toolkits showcase the benefits of OO programming, including the use of dynamic class composition. Our goal is, however, to expose the pitfalls of migrating untyped OO code to a typed world. To that end, we start with an untyped sample library for text editing widgets and then migrate some the code to an evolving explicitly typed dialect of the language.

Classes in Racket The editor%[2] class in figure 2 extends widget%, a base class provided by the GUI library. The new sub-class uses **super-new** to call the constructor of widget%. It defines three new public methods: on-key for handling key events, on-paint for drawing the contents to the screen, and save-file for saving the contents of the editor.

The first position in the class form, which specifies the superclass, is notably not limited to names of classes. We can supply *any* expression as long as it *evaluates* to a class. This feature allows the definition of runtime *mixins*.

Mixins A mixin is simply a function that take a class as an argument and subclasses it using the **class** form. With mixins, programmers may parameterize classes over base classes and thus decouple features from specific points in class hierarchies. The right-hand column in figure 2 presents a sample mixin that adds simple modal editing—as in `vi`—to an arbitrary editor.

[2] By convention, class names are suffixed with % in Racket.

Concretely, modal-mixin adds a mode field that controls which keybindings are active, initially set to the command mode. The mixin explicitly overrides the on-key method so that it dispatches key handling to the appropriate method.[3]

We can apply the mixin to create a concrete modal editor:

```
(define modal-editor% (modal-mixin editor%))
```

Not all applications of modal-mixin to base classes are successful. For example, if the base class already implements a toggle-mode method or lacks an on-key method, then mixin application fails with a runtime error.

In other words, **public** method definitions require the *absence* of the method in the superclass while **override** definitions require their *presence*. Our type system must express these restrictions in order to accommodate mixins or other dynamic patterns of composition.

As defined, modal-mixin can be applied to any class that lacks the method toggle-mode and has the method on-key. Thus, modal-mixin's functionality is not tied to a particular class hierarchy and is composable with other editor features. More concretely, we can compose modal-mixin with other mixins from our editor library to produce several combinations of functionality:

```
(define ide-editor%
  (modal-mixin
    (line-num-mixin (indent-mixin editor%))))
(define word-processing-editor%
  (modal-mixin
    (spelling-mixin (search-mixin editor%))))
```

In short, mixins are polymorphic with regard to the *specialization interface* [20] that the mixin expects. With this in mind, we need to consider how to accommodate the programming patterns we have just discussed in a typed setting.

2.2 A First Design for Typed First-Class Classes

From our previous discussion, we can identify two requirements for a typed sister language: it must support class polymorphism and its types must express constraints on method presence and absence. One possibility is to use a type system with bounded polymorphism and subtyping for class types. This choice is problematic, however, because runtime inheritance and subtyping are at odds.

For example, consider this code snippet:

```
(define typed-editor%
  (class widget%
    (super-new)
    (define/public (on-key [key : String]) : Void
      (case key
        [("C-x") (do-cut)] ...))
    (define/public (on-paint) : Void ... )
    (define/public (save-file [name : String]) : Void ... )
    (define/private (do-cut) : Void ... ) ... ))
```

[3] Like $C^\sharp$, Racket requires that **override** methods be explicitly identified.

It defines a typed variant of the editor% class from figure 2. The revised definition is like the untyped one but adds type annotations to methods. Width subtyping on class types allows casting a class to a narrower type, i.e., the type system can forget methods or fields. If the type system allowed width subtyping on class types, typed-editor% could be downcast to T = (**Class** ([on-paint **:** ($\rightarrow$ **Void**)])). This is fine for a client that uses the instances of class, as the instances would have the type (**Object** ([on-paint **:** ($\rightarrow$ **Void**)])) and therefore support only calls to the on-paint method. However, *inheritance* is now unsound because the type system has no knowledge of whether methods are actually present or absent.

Thus, the following definition type-checks but raises a runtime error because a public method is redefined:

```
(define piano%
  (class (cast T typed-editor%) ;; subsumption
    (super-new)
    ;; already defined in superclass
    (define/public (on-key [key : Note])
      : Void
      (play-sound key))))
```

In short, naive width subtyping for class types is unsound with respect to Racket's inheritance semantics; even for Java, the naive approach would fail because a subclass may override a method with a different type.

Row polymorphism Another possibility is to repurpose *row polymorphism* [36], studied in the context of type inference for objects [26, 36], object calculi [10], and extensible records [13, 18]. Functions over extensible records also require polymorphism over row members with constraints on the presence or absence of members. Unlike subtyping, row polymorphism prohibits forgetting class members. Instead, row polymorphism for classes abstracts over the features of a class type—both method signatures and types of fields—using constraints on row variables to express absence and row signatures to express presence.

With row polymorphic types, the type of modal-mixin from figure 2 could be written as follows:

```
(∀ (r / on-key toggle-mode))
  ((Class ([on-key : (String → Void)] | r))
   →
   (Class ([toggle-mode : (→ Void)]
          [on-key : (String → Void)] | r))))])
```

The mixin's type is given as a function type that is polymorphic in a row variable r, which, in turn, is constrained to lack on-key and toggle-mode members. These particular constraints are needed because r is a placeholder for the rest of the methods or fields, which should contain neither on-key nor toggle-mode. If r has an on-key method, it conflicts with on-key in the argument class that the mixin overrides. Since

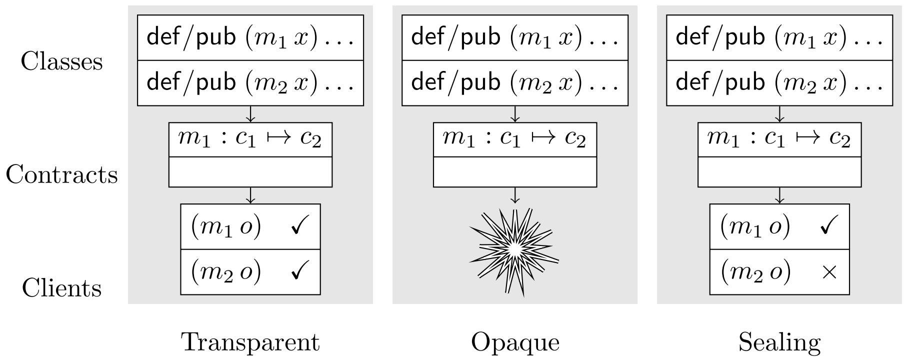

Figure 3. Kinds of class contracts

modal-mixin adds toggle-mode to its base class, whose type contains r, the row variable r must also lack toggle-mode.

Row variables are instantiated like ordinary type variables. Applying a row abstraction to a concrete row type requires that the row satisfies the constraints on the variable. Thus, the variable r above can be instantiated with the concrete row ([on-paint **:** (→ **Void**)] [insert **:** (**String** → **Void**)]) but not with the row ([on-key **:** (**String** → **Void**)]) or a row that contains this signature, due to the constraints on on-key.

By using row polymorphism, we make a tradeoff with subtyping. Class abstractions specify polymorphism explicitly in their type, rather than allowing the use of the subsumption rule. That is, we shift the burden of typing to the *implementor* of a mixin. This does not complicate the application of mixins because, in practice, type application of rows can be inferred [35]. In contrast, our type system *does* allow width subtyping on the types of objects, since objects in Racket cannot be extended after creation.

In other words, our approach is pay-as-you-go. Programmers need to be aware of row polymorphism only when dealing with abstractions that are not provided in most typed object-oriented languages, such as mixins. Meanwhile, programmers need not consider row polymorphism when dealing with objects, and can instead reason about their programs using familiar subtyping rules.

2.3 Protecting Typed Code from Untyped Code

Types are only half of the gradual typing story. Runtime monitoring of type system invariants is the other half, which we handle with higher-order contracts [9, 29] in the style of Tobin-Hochstadt and Felleisen [32]. Every type for a class becomes a run-time contract at the boundary where it flows from a typed module to an untyped module or vice versa. A contract for first-class classes specifies clauses that protect methods and fields of the contracted class. Class contracts are specified separately from the definition of the class, since classes are values and are not defined in a static hierarchy.

Our contract-based approach does not scale smoothly because we wish to formulate types that are polymorphic in method and field clauses. Figure 3 presents illustrative diagrams for three different ways of dealing with methods or fields not included in a class contract. Strickland and Felleisen's contracts [29] are *transparent*: they allow unmentioned methods or fields to pass through without any enforcement. Our design requires two additional mechanisms for handling unmentioned behavior in order to fully protect typed code: *opaque* class contracts and *sealing* class contracts. *Opaque* contracts disallow any unmentioned behavior by raising a contract error at the boundary. *Sealing* contracts allow unmentioned behavior to pass through the boundary but require a matching unsealing to access it.

```
"typed-editor.rkt"

import editor% from "editor.rkt" with type:
  (Class [on-key (String → Void)]
         [on-paint (→ Void)] ...)

(send (new editor%) on-key "C-x")
```

Opaque contracts Suppose we import the editor functionality from the untyped module in figure 2 into the nearby typed module typed-editor.rkt. Note that we use a double border to distinguish typed modules from untyped ones. Since the editor class is untyped, we must specify an import type for the class so that Typed Racket can type check its uses. The given class type specifies the signatures of the individual methods. To protect this import, a natural choice would be to translate the type into a class contract that checks matching predicates. Unfortunately, the existing class contracts implement *transparent* behavior, i.e., they require that the methods mentioned in the contract have the specified behaviors but make no guarantees about any unspecified methods.

Hence, transparent contracts cannot protect typed code properly, even in this simple case. Recall that editor% in figure 2 implements a method that is absent from its import type, namely save-file, and thus its corresponding transparent import contract allows the method to pass through. In the code fragment labeled "typed-editor.rkt" (continued), this would result in a name conflict in the typed module because the type system does not know that editor% provides save-file. That is, the code would type-check but signal a runtime error blaming the typed module for an error that the type system seemingly ruled out.

"typed-editor.rkt" (continued)

```
...
(define my-editor%
  (class editor%
    (super-new)
    ;; error: save-file must be absent in editor%
    (define/public (save-file [filename : String])
       : Void
       ...)))
```

To avoid this safety gap, we introduce and use opaque contracts so that typed code cannot accidentally import classes that contain unknown methods. Note that the introduction of opaque contracts and the prohibition on subtyping for class types are due to the same underlying cause, namely the desire to retain familiar inheritance semantics.

"typed-modal.rkt"

```
import editor% from "typed-editor.rkt"
import modal-mixin from "editor.rkt" with type:
  (∀ (r / on-key toggle-mode)
    ((Class ([on-key : (String → Void)] | r))
     →
     (Class ([toggle-mode : (→ Void)]
             [on-key : (String → Void)] | r))))

(modal-mixin% editor%)
```

Sealing contracts For every feature in our type system, we need a corresponding feature in the contract system to enforce its invariants at runtime. Thus, we need some form of parametric contract to protect row polymorphic functions on classes. Consider what happens when an untyped mixin is imported into the typed module "typed-modal.rkt" below. This module imports modal-mixin from figure 2 with a parametric type and calls the mixin on the editor% class from "typed-editor.rkt". Assuming modal-mixin is defined properly, no runtime error can blame the typed module.

Suppose, however, that the programmer who wrote "editor.rkt" adds another public method to modal-mixin, as in the revised version of "editor.rkt". Even if the creator of typed-modal.rkt does not adapt the type to the new situation, our system must discover the additional method and signal a violation—otherwise the type system would be unsound. One apparent option is to map the type to an opaque contract, which prohibits the flow of modal-mixin across the module boundary. Unfortunately, opaque contracts would also rule out the application of the correct version of modal-mixin to a base class with methods other than on-key.

"editor.rkt" (revised)

```
...
(define (modal-mixin base%)
  (class base%
    (super-new)
    ... ;; as above
    (define/public (switch-mode) ... )
    ...))
```

Instead, we introduce and use *sealing contracts*, which prevent the addition or use of sealed methods until a corresponding unsealing contract is applied. Sealing contracts use unforgeable keys generated at run-time to prevent unauthorized access. This prevents modal-mixin from adding unspecified methods but allows unspecified methods from the base class to flow through mixin application. In other words, sealing contracts for mixin types reflect their key feature, polymorphism over base types.

Metaphorically speaking, sealing contracts establish a private channel through one component to another. We use such a channel to send a typed class through an untyped mixin, ensuring that the mixin cannot tamper with the protected names of the class en route. A sealing contract at a negative position (e.g., function argument) establishes an entrance to the channel. Dually, a contract in a positive position (e.g., function result) establishes an exit. The ends of the channel are locked with unforgeable keys, allowing only authorized code to send and receive values on the channel.

Our system seals classes because class types are polymorphic. Classes that pass through sealing contracts are not completely inaccessible, however. Instead, sealing is applied at the granularity of class members such as methods and fields. Attempts to invoke a sealed method name, access a sealed field name, or extend a class using a sealed name all fail with a contract error. In contrast, sealing contracts do not impose any access limitations on exposed methods. This establishes the connection with row polymorphism: class types allow the use of concrete members at the specified type but disallow the use of abstract members from a row variable.

Like most OO languages, Racket also contains stateful operations. In general, the combination of polymorphic contracts and state requires some care to ensure soundness. In particular, the precise timing for key generation on seals is crucial. One choice is to generate keys for seals when a contract is applied, e.g., when an untyped value is imported into a typed module. For example, consider the situation in figure 4a. The "state.rkt" module exports state-mixin and "broken-client.rkt" imports the mixin with a type that suggests it is the identity function on classes. Suppose that keys for seals are generated once as the mixin is imported. The first time that state-mixin is called, editor% is sealed on en-

```
"state.rkt"
(define (state-mixin base-class)
  (define storage #f)
  (cond [(not storage)
         (set! storage base-class)
         base-class]
        [else storage]))
```

```
"broken-client.rkt"
import state-mixin from "state.rkt"
  with type:
  (∀ (r) (Class (| r)) → (Class (| r)))

(send (new (state-mixin editor%))
      on-key)
(send (new (state-mixin modal-editor%))
      toggle-mode)
```

(a) Misuse of state

```
"search.rkt"
(define (search-mixin base-class)
  (class base-class
    (super-new)
    (define/public (search string)
      . . .
      ;; bad call to save-file
      (send this save-file 0))))
```

```
"search-client.rkt"
import editor% from "typed-editor.rkt"
import search-mixin% from "search.rkt"
  with type:
  (∀ (r / autosave)
    (→ (Class ( | r))
       (Class
         (search : (String → Void) | r))))

(define searchable% (search-mixin editor%))
(send (new searchable%) search "x")
```

(b) Invalid method invocation

Figure 4. Potential violations of soundness via mixins

try to the mixin and unsealed on exit. Sending the on-key message is thus acceptable, and no contract errors are raised.

The next line is a call to state-mixin, perhaps formulated under the assumption that it behaves like the identity function—a possibility suggested by its row-polymorphic type. This time, the mixin is applied to modal-editor%, which is sealed with the same key as editor%. This allows state-mixin to return editor% without triggering a contract error, and that cause the method missing error for toggle-mode. In short, even though the code type-checked in our imaginary type system, it caused a run-time type error.

To be sound, our system must generate these seals at each mixin *invocation*. This ensures that editor% and modal-editor% are sealed with distinct unforgeable keys and that the second call to state-mixin signals a contract error.

Object contracts are like class contracts, except that they transparently protect particular instances. Since object types offer width subtyping, contracts on objects need not enforce the kind of opacity that class contracts enforce.

2.4 Access restrictions for method invocation

While opaque contracts ensures that untyped code cannot access methods unlisted in interfaces, dynamic uses of inheritance (e.g., mixins or traits) create a potential hole in this protection. Consider the example in figure 4b. An untyped mixin imported into the "search-client.rkt" module has a method search that assumes the existence of a save-file method in its base class. Moreover, search provides a faulty argument to save-file. When search is invoked on an instance of searchable%, an error occurs due to that faulty argument. Since the type given in the client module does not mention

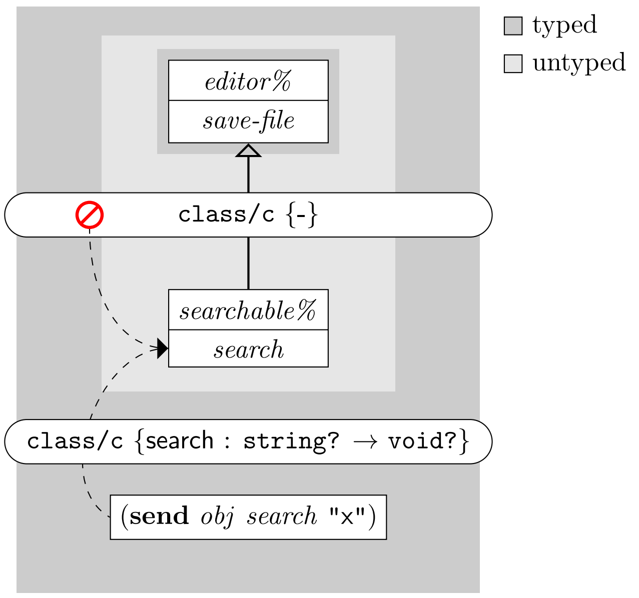

Figure 5. A snapshot of the class hierarchy after unsealing

$$
\begin{array}{rcll}
e & ::= & v \mid x \mid (e\ e) \mid op(e) \mid \texttt{if}\ e\ e\ e \mid \texttt{send}(e, m, e) \mid \texttt{iget}(f^{*}, e) \mid \texttt{iset!}(f^{*}, e, e) \mid \texttt{new}(e) & \textit{(expressions)} \\
 & & \mid \texttt{mon}_{l}^{l,l}(c, e) \mid \texttt{class}(e)\,\{\overline{f := v}\ \ \overline{m^{p}x\,.\,e}\ \ \overline{m^{o}x\,.\,e}\} & \\
op & ::= & \texttt{num?} \mid \texttt{bool?} & \textit{(primitives)} \\
v & ::= & \texttt{\#t} \mid \texttt{\#f} \mid n \mid cv \mid o \mid \lambda x\,.\,e & \textit{(values)} \\
cv & ::= & \texttt{object\%} \mid \texttt{class/v}^{\iota}(cv)\,\{\overline{(f, v)}\ \ \overline{mx\,.\,e}\} & \textit{(class values)} \\
o & ::= & \texttt{object/v}(cv)\,\{\overline{(f, a)}\} & \textit{(objects)} \\
f^{*} & ::= & f \mid a & \textit{(field terms)} \\
x & \in & \text{Var} & \textit{(variables)} \\
f & \in & \text{Field} & \textit{(fields)} \\
m & \in & \text{Method} & \textit{(method names)} \\
a & \in & \text{Location} & \textit{(locations)} \\
l, j, k & \in & \text{Label} & \textit{(contract labels)}
\end{array}
$$

Figure 6. Untyped Expressions

save-file and the class is unsealed by the time it is instantiated, sealing contracts do not catch this faulty method invocation. That is, a method invocation from untyped code to typed code can be unsafe and must be rejected unless a contract protects the method.

To prevent faulty access via mixins, we require method calls to dispatch to either a method defined within the same component or a contracted method. That is, if untyped code invokes a method that is not protected by a contract, its definition must reside in untyped code as well. The same condition applies to typed code.

Figure 5 illustrates this restriction graphically. The diagram shows the object interactions established by the code in figure 4b. There are two relevant component transitions here: one between the editor% class and the mixin from the untyped component, and one between searchable%—created by the mixin—and its use in the typed component. A contract protects both boundaries, but the inner contract disallows the call to save-file because the contract specification does not include that method.

The problem with cross-boundaries access was discovered during a first, failed attempt to establish the soundness of our type system design. Conversely, the failure suggested this constraint on protected method calls. Fortunately, this constraint does not reduce the expressiveness of purely typed or untyped code, but it requires that invocations that transfer control from typed to untyped code (and vice versa) are properly monitored by the contract system.

3. Formalizing Typed First-Class Classes

Our examples have exposed thorny problems about programs that use dynamic class composition across the boundary between typed and untyped components. To communicate our solution, we present a formal model of gradually-typed dynamic class composition. The model is both a vehicle for a compact presentation of our design and a platform for proving its soundness. We consider type soundness the baseline property of any type system, to be preserved even when typed components are linked with untyped code.

3.1 Syntax

Our language, called TFCC, allows the embedding of typed terms in untyped terms and vice versa. The interactions between them are mediated by monitors with contracts. A monitored term can be thought of as a server module that exports services to its context, i.e., its client(s). This section starts with a look at the untyped portion of the language (figure 6) and then proceeds to the typed portion (figure 7).

Expressions include values, variables, applications, primitive operations, conditionals, method invocation, object instantiation, field mutation, contract monitors, and classes. The set of ***values*** includes booleans, numbers, class values, objects, and λ-abstractions. Method invocations are written $\texttt{send}(e_0, m, e_1)$ where e_0 is the receiver, m the method name, and e_1 the argument to the method.

The expression $\texttt{class}(e)\,\{\overline{f := v}\ \ \overline{m^{p}x\,.\,e^{p}}\ \ \overline{m^{o}x\,.\,e^{o}}\}$ consists of a superclass expression e, field names $\overline{f}$ paired with corresponding initial values, and definitions of **public** methods $\overline{m^{p}}$ and **override** methods $\overline{m^{o}}$. If the superclass expression evaluates to a suitable value, a class reduces to a class value class/v with a unique key ι that is used for method dispatch. The term object% represents the root of the class hierarchy. Field declarations produce mutable local fields, which are only accessible from within method bodies. We include state so that the calculus can express examples that introduce potential unsoundness in the absence of appropriate dynamic sealing, as explained in section 2.

State Fields make objects stateful. The iget and iset! expressions, respectively, get and set field values locally.

Types include **Int** and **Bool** plus function types. An object or class type consists of rows, describing method signatures.

$$
\begin{array}{rcll}
\sigma & ::= & \tau \mid \forall(\rho\backslash\overline{m}).\tau & \text{(type schemes)}\\
\tau & ::= & \textbf{Int} \mid \textbf{Bool} \mid \textbf{Class}\ er \mid \textbf{Object}\ r \mid \tau \to \tau & \text{(types)}\\
r & ::= & \{\overline{(m : \tau)}\} & \text{(rows)}\\
er & ::= & r \mid \{\overline{(m : \tau)} \mid \rho\} & \text{(extended rows)}
\end{array}
$$

$$
\begin{array}{rcll}
e_\tau & ::= & (e_\tau\ e_\tau) \mid x \mid op(e_\tau) \mid \texttt{if}\ e_\tau\ e_\tau\ e_\tau \mid \Lambda(\rho\backslash\overline{m}).\, e_\tau \mid e_\tau\ [r] \mid \texttt{send}(e_\tau,\, m,\, e_\tau) & \text{(typed expressions)}\\
 & \mid & \texttt{iget}(f^*,\, e_\tau) \mid \texttt{iset!}(f^*,\, e_\tau,\, e_\tau) \mid \texttt{new}(e_\tau) \mid \texttt{mon}_l^{l,l}(c,\, e) & \\
 & \mid & \texttt{class}(e_\tau)\, \{\overline{f : \tau := v_\tau}\ \ \overline{m^p(x : \tau) : \tau\, e_\tau}\ \ \overline{m^o(x : \tau) : \tau\, e_\tau}\} & \\
v_\tau & ::= & \texttt{\#t} \mid \texttt{\#f} \mid \texttt{n} \mid cv_\tau \mid o_\tau \mid \lambda(x : \tau)\, e_\tau & \text{(typed values)}\\
cv & ::= & \texttt{object\%} \mid \texttt{class/v}^\iota(cv_\tau)\, \{\overline{(f, \tau, v_\tau)}\ \ \overline{m^p(x : \tau) : \tau\, e_\tau}\ \ \overline{m^o(x : \tau) : \tau\, e_\tau}\} & \text{(typed class values)}\\
o & ::= & \texttt{object/v}(cv_\tau)\, \{\overline{(f, \tau, a)}\} & \text{(typed objects)}
\end{array}
$$

Figure 7. Typed Expressions

For example, $\{(\text{on-key} : \tau_1), (\text{on-paint} : \tau_2)\}$ is a row with labels on-key and on-paint and types τ_1 and τ_2. Rows in class types may be extended with a *row variable* as in $\{(\text{on-key} : \tau_1), (\text{on-paint} : \tau_2) \mid \rho\}$. With $r_1 \oplus r_2$, we concatenate two rows.

Typed expressions require type annotations for all variable declarations. Since we abstract over rows, we also have type abstractions $\Lambda(\rho\backslash\overline{m}).\, e_\tau$ and type applications $e_\tau\ [r]$.

Untyped-Typed Interaction A monitor $\texttt{mon}_j^{k,l}(c,\, e)$ separates a program into components [5], i.e., e and the context of mon, mediated by the contract c. The superscript labels k and l name the *server* and *client* components respectively. The label j names the contract c. In our model, monitors play only one role. They mediate between typed and untyped components. Hence it suffices to use just two labels, instead of unique labels per components: u for untyped, t for typed.

Our reduction semantics models exchanges of values between components with substitutions of monitored values, which embeds typed values within untyped code and vice versa. A typed-in-untyped embedding is valid if the monitor's contract is related to the type of the embedded term; an untyped-in-typed embedding is valid if it type-checks in the typed context with the contract interpreted as a type. The bijective function $\mathsf{T}[\![]\!]$, defined in figure 9, specifies the natural correspondence between contracts and types.

Contracts check type-like properties at runtime. In contrast to Eiffel, our higher-order contracts describe the behaviors of entire objects, including their methods [9, 29].

The model's contract language includes flat or predicate contracts, function contracts, class contracts, and contracts for parametric functions. Figure 8 presents the syntax of contracts in two parts: *pre-contracts* are surface syntax, which are elaborated into *(core) contracts*.

$$
\begin{array}{rcll}
c & ::= & \texttt{flat}(op) \mid c \mapsto c & \text{(pre-contracts)}\\
 & \mid & \forall^c(\rho\backslash\overline{m}).(c \mapsto c) \mid \texttt{class/c}^\bullet(\overline{[m\ c \mapsto c]}) & \\
 & \mid & \texttt{class/c}^*(\rho, \overline{[m\ c \mapsto c]}) \mid \texttt{object/c}(\overline{[m\ c \mapsto c]}) &
\end{array}
$$

$$
\begin{array}{rcll}
e & ::= & \ldots \mid \texttt{blame}_l^l & \text{(expressions)}\\
c & ::= & \texttt{flat}(op) \mid c \mapsto c & \text{(core contracts)}\\
 & \mid & \forall^c(\rho\backslash\overline{m}).(c \mapsto c) \mid \texttt{class/c}^\bullet(\overline{[m\ c \mapsto c]}) & \\
 & \mid & \texttt{seal/c}(\overline{[m\ c \mapsto c]}, \overline{m}, \gamma) & \\
 & \mid & \texttt{unseal/c}(\overline{[m\ c \mapsto c]}, \gamma) \mid \texttt{object/c}(\overline{[m\ c \mapsto c]}) & \\
\gamma & ::= & \varsigma \mid \rho & \text{(key terms)}\\
\varsigma & \in & \text{Key} & \text{(keys)}
\end{array}
$$

$$
\begin{array}{rcll}
v & ::= & \ldots \mid \forall\texttt{G}_l^{l,l}(\rho\backslash\overline{m}).(c \mapsto c)\{v\} & \text{(values)}\\
cv & ::= & \ldots \mid \texttt{G}_l^{l,l}\{cv, (\overline{m\ c \mapsto c})\} & \text{(class values)}\\
 & \mid & \texttt{SG}_l^{l,l}\{cv, \overline{m}, \varsigma\} & \\
o & ::= & \ldots \mid \texttt{OG}_j^{k,l}\{o, \overline{[m\ c \mapsto c]}\} & \text{(objects)}
\end{array}
$$

Figure 8. Contracts and guards

Class contracts come in two varieties: *opaque* and *sealing*. The former differs from *transparent* class contracts [29] in that they enforce the absence of methods not mentioned in a contract. This feature of $\texttt{class/c}^\bullet$ ensures that typed modules can safely import classes from untyped modules.

Sealing contracts $\texttt{class/c}^*$ have meaning only within a row polymorphic contract. They are used to specify whether a given contract position should be polymorphic. For a negative position, an elaboration from pre-contracts to core contracts translates $\texttt{class/c}^*$ to a *sealing* contract seal/c; it becomes an unseal/c contract in a positive position. Both seal/c and unseal/c contain γ, which is either a variable or a *key* for unlocking seals.[4]

[4] The variable case is required by our choice of semantics and occurs only in intermediate reduction steps.

The runtime syntax also includes guards. Guards act like contract monitors but, unlike monitors, are values. Since guards are values, they can pass through contract boundaries. The parametric guard $\forall\mathtt{G}_j^{k,l}(\rho\backslash\overline{m}).(c \mapsto c)\{v\}$ behaves like a function. When applied, it generates a fresh seal key for its contract. This ensures that keys cannot be forged using state, as explained in section 2.3.

Since contract checking for a class is delayed until its methods are invoked, we use $\mathtt{G}_j^{k,l}\{cv, (\overline{m\ c \mapsto c})\}$ to retain the contracts in the class hierarchy. Similarly, the sealing guard $\mathtt{SG}_j^{l,k}\{cv, \overline{[m\ c_1 \mapsto c_2]}, \overline{m'}, \varsigma\}$ wraps a class cv in order to retain seals until they are checked.

$$
\begin{aligned}
\mathsf{T}[\![\mathtt{flat(int?)}]\!] &= \textbf{Int} \\
\mathsf{T}[\![\mathtt{flat(bool?)}]\!] &= \textbf{Bool} \\
\mathsf{T}[\![c_1 \mapsto c_2]\!] &= \mathsf{T}[\![c_1]\!] \rightarrow \mathsf{T}[\![c_2]\!] \\
\mathsf{T}[\![\forall^{\mathtt{c}}(\rho\backslash\overline{m}).(c_1 \mapsto c_2)]\!] &= \forall\rho\backslash\overline{m}.\mathsf{T}[\![c_1 \mapsto c_2]\!] \\
\mathsf{T}[\![\mathtt{class/c}^{\bullet}(\overline{[m\ c_1 \mapsto c_2]})]\!] &= \\
&\textbf{Class}\,\{\overline{(m : \mathsf{T}[\![c_1 \mapsto c_2]\!])}\} \\
\mathsf{T}[\![\mathtt{seal/c}(\overline{[m\ c_1 \mapsto c_2]}, \overline{m}, \rho)]\!] &= \\
&\textbf{Class}\,\{\overline{(m : \mathsf{T}[\![c_1 \mapsto c_2]\!])} \mid \rho\} \\
\mathsf{T}[\![\mathtt{unseal/c}(\overline{[m\ c_1 \mapsto c_2]}, \rho)]\!] &= \\
&\textbf{Class}\,\{\overline{(m : \mathsf{T}[\![c_1 \mapsto c_2]\!])} \mid \rho\} \\
\mathsf{T}[\![\mathtt{object/c}(\overline{[m\ c_1 \mapsto c_2]})]\!] &= \\
&\textbf{Object}\,\{\overline{(m : \mathsf{T}[\![c_1 \mapsto c_2]\!])}\}
\end{aligned}
$$

Figure 9. Type-contract correspondence

3.2 Type System

The type system is based on a typed λ-calculus with subtyping, limited mutable variables, object types, and class types. The important typing rules are shown in figure 10. The rules use a judgement $\Gamma \,|\, \Sigma \vdash e : \tau$ that states that a term e has type τ assuming free variables are typed in Γ and store locations in Σ. Store typings are used to type-check operations on the private fields of objects.

Row abstractions are checked using T-RowAbs. Applications of abstractions to rows (T-RowApp) require that the provided row matches the absence labels on the abstraction's bound variable using the $\Gamma \vdash \rho\backslash\overline{m}$ judgement.

The interesting typing rules are those involving class and object types. A class is well-typed (T-Class) if its superclass is a valid class and allows extension with the new **public** and **override** methods. These conditions are ensured by the judgements that a row lacks a member, $\Gamma \vdash er\backslash m$, and that a row has a member, $\Gamma \vdash m \in er$. If the superclass type contains a row variable, then all methods must be compatible with the absence labels on the variable. Fields and methods must all be well-typed in the usual sense. Method bodies are checked under the assumption that the receiver has type **Object** $\mathsf{R}[\![er]\!]$, where $\mathsf{R}[\![er]\!]$ denotes the class's row but *without* the row variable if er contains one.

Object instantiation (T-New) requires that the instantiated class has a concrete type, i.e., any type variables have been instantiated. Method invocation is checked by T-Send.

Since objects have concrete rows, an object type is a subtype of another if the corresponding rows are subtypes via the standard width subtyping rule.[5] Meanwhile, there are no subtyping rules for classes because row polymorphism replaces subtyping for classes, as detailed in section 2.

We defer rules for interactions between typed and untyped code to section 4.3, where we explain the soundness theorem for mixed type programs.

3.3 Operational Semantics

Our operational semantics uses context-sensitive reduction rules [7]. Figure 12 presents the evaluation contexts for our language and lists the reduction rules, which in turn rely on the metafunctions defined in figure 13. The reflexive-transitive closure of the reductions determines the evaluation function.

We first explain the semantics of the base language without contracts and then incrementally introduce the cases for contract monitoring. The typed language has the exact same semantics as the untyped one. For some purposes, we assume that the types are first stripped from typed expressions; in other cases, we assume that the reduction system carries type information without using it.

The reduction rules in figure 11 define the conventional behavior of λ-expressions, primitives, and conditionals; we omit the obvious definition of δ. The semantics of first-class classes is straightforward as well. The evaluation contexts ensure that class expressions reduce only after the superclass expressions are reduced to values. A class successfully reduces to a value (Class) when all of its **override** methods are present in the superclass and its **public** methods absent.

Method invocation (Send) triggers only if the given method m is present in the class hierarchy of the receiving object. The rule relies on a metafunction to look up the target method in the class hierarchy of the receiver object. The Pull function traverses the hierarchy to locate the method.

The rules for state are conventional. Creating a new object from a class (New) chooses unallocated addresses in the store and reduces to an object value. Similarly, getting and setting fields (Get, Set) just involves looking up and replacing values in the store.

Figure 12 introduces contract monitoring. Since reductions may result in contract violations, reductions may produce the error term $\mathtt{blame}_j^l$ that pinpoints the misbehaving

[5] Depth subtyping is omitted for simplicity, but we conjecture that its addition poses no problems.

$$\boxed{\Gamma \,|\, \Sigma \vdash e : \tau}$$

$$\textsc{T-RowAbs}\quad \frac{\Gamma, (\rho\backslash\overline{m}) \,|\, \Sigma \vdash e : \tau}{\Gamma \,|\, \Sigma \vdash \Lambda(\rho\backslash\overline{m}).\, e : \forall(\rho\backslash\overline{m}).\tau} \qquad \textsc{T-RowApp}\quad \frac{\Gamma \,|\, \Sigma \vdash e : \forall(\rho\backslash\overline{m}).\tau \qquad \Gamma \vdash r\backslash\overline{m}}{\Gamma \,|\, \Sigma \vdash e\ [r] : [r/\rho]\,\tau}$$

$$\textsc{T-Send}\quad \frac{\begin{array}{c}\Gamma \,|\, \Sigma \vdash e_0 : \textbf{Object}\ r \\ (m : \tau_1 \to \tau_2) \in r \qquad \Gamma \,|\, \Sigma \vdash e_1 : \tau_3 \qquad \tau_3 <: \tau_1\end{array}}{\Gamma \,|\, \Sigma \vdash \texttt{send}(e_0,\, m,\, e_1) : \tau_2} \qquad \textsc{T-New}\quad \frac{\Gamma \,|\, \Sigma \vdash e : \textbf{Class}\ r}{\Gamma \,|\, \Sigma \vdash \texttt{new}(e) : \textbf{Object}\ r} \qquad \textsc{T-Root}\quad \Gamma \,|\, \Sigma \vdash \texttt{object\%} : \textbf{Class}\,\{\}$$

$$\textsc{T-Object}\quad \frac{\Gamma \,|\, \Sigma \vdash cv : \textbf{Class}\ r \qquad \forall i, \Sigma(a_i) = \tau_i}{\Gamma \,|\, \Sigma \vdash \texttt{object/v}(cv)\,\{\overline{(f_i, \tau_i, a_i)}\} : \textbf{Object}\ r}$$

$$\textsc{T-Class}\quad \frac{\begin{array}{c}\Gamma \,|\, \Sigma \vdash e_s : \textbf{Class}\ er_s \qquad \Gamma \vdash \overline{er_s\backslash m_i^p} \qquad \Gamma \vdash m_i^o \in er_s \\ \Gamma, \texttt{this} : \tau_{obj}, x_i^p : \tau_{1_i}^p \overline{f : \tau_f} \,|\, \Sigma \vdash e_i^p : \tau_{2_i}^p \\ \Gamma, \texttt{this} : \tau_{obj}, x_i^o : \tau_{1_i}^o, \overline{f : \tau_f} \,|\, \Sigma \vdash e_i^o : \tau_{2_i}^o \\ \Gamma \vdash v_{f_i} : \tau_{f_i} \qquad er = er_s \oplus \{\overline{(m^p : \tau_1^p \to \tau_2^p)}\} \qquad \tau_{obj} = \textbf{Object}\ \mathsf{R}[\![er]\!]\end{array}}{\Gamma \vdash \texttt{class}(e_s)\,\{\overline{f : \tau_f := v_f}\ \ \overline{m^p(x^p : \tau_1^p) : \tau_2^p\, e^p}\ \ \overline{m^o(x^o : \tau_1^o) : \tau_2^o\, e^o}\} : \textbf{Class}\ er}$$

Figure 10. Selected type rules

$$\begin{array}{l}E ::= [\,] \mid (E\ e) \mid (v\ E) \mid op(E) \mid \texttt{if}\ E\ e\ e \mid \texttt{send}(E,\, m,\, e) \mid \texttt{send}(v,\, m,\, E) \mid \texttt{iget}^\iota(f^*,\, E) \\ \qquad \mid \texttt{iset!}^\iota(f^*,\, E,\, e) \mid \texttt{iset!}^\iota(f^*,\, e,\, E) \mid \texttt{new}(E) \mid \texttt{mon}_l^{l,l}(c,\, E) \mid \texttt{class}(E)\,\{\overline{f := v}\ \ \overline{mx.e}\ \ \overline{mx.e}\}\end{array}$$

$$\begin{array}{llll}
\langle E[\cdots], S\rangle & \hookrightarrow & \langle E[\cdots], S\rangle & \\
\hline
(\lambda x.\, e\ \ v) & \cdot & [v/x]\, e & \textsc{Beta} \\
op(v) & \cdot & \delta(op, v) & \textsc{Delta} \\
\texttt{if #t}\ e_2\ e_3 & \cdot & e_2 & \textsc{IfTrue} \\
\texttt{if #f}\ e_2\ e_3 & \cdot & e_3 & \textsc{IfFalse} \\
\hline
\texttt{class}(cv)\,\{\overline{f := v}\ \ \overline{m^p x^p.\, e^p}\ \ \overline{m^o x^o.\, e^o}\} & . & \texttt{class/v}^\iota(cv)\,\{\overline{(f, v)}\ \ \overline{mx.e}\} & \textsc{Class} \\
\quad \text{if } \overline{m^p} \cap \mathsf{Methods}(cv) = \emptyset \text{ and } \overline{m^o} \subseteq \mathsf{Exposed}(cv) & & & \\
\quad \text{and where } \iota \text{ is fresh and } \overline{mx.e} = \overline{m^p x^p.\, [\Box/^{id}\iota]\, e^p} \oplus \overline{m^o x^o.\, [\Box/^{id}\iota]\, e^o} & & & \\
\texttt{send}(o,\, m,\, v) & . & (\left[\texttt{this}/\, e_o, \overline{a/f}\right] e\ \ v) & \textsc{Send} \\
\quad \text{if } m \in \mathsf{Methods}(o) & & & \\
\quad \text{and where } e = \mathsf{Pull}(o, m) \text{ and } \overline{(f, a, \iota)} = \mathsf{ObjectFields}(o) & & & \\
\hline
\langle E[\texttt{new}(cv)], S\rangle & \cdot & \langle E[\texttt{object/v}(cv)\,\{\overline{(f, a, \iota)}\}], [\overline{v}/\overline{a}]\, S\rangle & \textsc{New} \\
\quad \text{where } \overline{(f, v, \iota)} = \mathsf{Fields}(cv) \text{ with } \overline{a} \cap dom(S) = \emptyset & & & \\
\langle E[\texttt{iget}^\iota(a,\, o)], S\rangle & \cdot & \langle E[v], S\rangle & \textsc{Get} \\
\quad \text{if } (f, a, \iota) \in \mathsf{ObjectFields}(o) \text{ and where } v = S(a) & & & \\
\langle E[\texttt{iset!}^\iota(a,\, o,\, v)], S\rangle & \cdot & \langle E[v], S\,[a/v]\rangle & \textsc{Set} \\
\quad \text{if } (f, a, \iota) \in \mathsf{ObjectFields}(o) & & &
\end{array}$$

Figure 11. Reductions for the base language

component l as violating contract j. Blame is propagated through all evaluation contexts.

Contracts control how monitors behave. Depending on the kind of contract, monitors reduce to additional monitors or guards, which immediately check the contract or delay checking in higher-order cases. When immediate checks fail, contracted values reduce to contract errors, blaming a specific component l for violating some fixed contract j.

The reduction rules for monitors deserve special attention. Monitoring an immediate contract reduces to a conditional expression (FLATC). If the predicate fails, a contract error is signalled. A monitor for a function contract reduces to a wrapped function (FUNC), which monitors argument and result contracts with appropriate blame [9].

Monitors with class or object contracts reduce to guards that wrap the appropriate classes or objects for higher-order contract checking [29]. Since our class contracts are opaque, they enforce the additional constraint that the contracted methods are the actual methods of the protected class (CLASSC). This constraint protects against the invocation of unspecified methods, which would otherwise violate safety as illustrated with the "typed-editor.rkt" example in section 2.3. Similarly, object contracts ensure that the protected object actually has the contracted methods (OBJECTC).

When monitors create guards, the latter track the contracts that need to be carried along for method invocation. Thus, the Pull metafunction delivers more than a method definition. When looking up a method, Pull attaches the appropriate contracts to the method as they are discovered in the class hierarchy. If Pull succeeds, the call reduces to the wrapped expression applied to the argument value.

Two other details are necessary to understand method invocation. The receiver of a method call (i.e., `this`) is wrapped in contracts found between the method definition and the caller by ProtectThis. Doing so ensures that dynamically dispatched calls are protected with the expected contracts. Furthermore, the semantics also ensures that methods can be invoked only if both the caller and method are in the same component. This prevents a mixin from invoking a method without protection, preventing the faulty invocation from the search-mixin example in section 2.3. In other words, all method invocation must either be dispatched locally or through a contract monitor.

The remaining contract forms track sealing and unsealing of methods in order to ensure sound interactions between typed and untyped code in the presence of functions that manipulate classes. Such a function must be monitored by a parametric contract, which reduces to a parametric guard ($\forall$C). The guard behaves like a function, but with additional monitoring ($\forall$G-APP). An application of the guard results in the generation of a seal key ς. It is crucial, as we demonstrated through the broken-mixin in subsection 2.3, that every application is sealed separately in order to prevent violations of row parametricity due to state.

The contracts contained in a parametric contract are either sealing or unsealing contracts. A sealing contract checks that all of the contracted methods are present in the class and reduces to a seal guard. Similarly, an unseal contract reduces to an unsealed class protected with a guard.

The semantics of sealing requires additional side conditions for the class and method invocation cases. For class expressions, the semantics requires that none of the **public** or **override** methods are sealed in the superclass.

Let us illustrate the operational semantics of sealing with a sample reduction sequence based on the example in figure 4b. In that example, a typed program invokes the search method on an object whose class was constructed by the application of an untyped but monitored mixin:

$$\begin{array}{lcl}
c & = & \mathtt{mon}_j^{u,t}(\mathit{ctc},\ \mathit{mixin})\ \mathit{editor\%} \\
e & = & \mathtt{send}(\mathtt{new}(c), s, \mathtt{"x"}) \\
\mathit{mixin} & = & \lambda c.\ \mathtt{class}(c)\{s\,x.\,\mathtt{send}(\mathtt{this},\ \mathtt{sf},\ 0)\ -\}) \\
\mathit{ctc} & = & \forall^{\mathbf{c}}(\rho\backslash -).(\mathtt{seal/c}([-]\,, \emptyset, \rho)) \\
 & & \qquad\qquad \mapsto \\
 & & \qquad\qquad \mathtt{unseal/c}([s\ \mathit{sctc}]\,, \rho) \\
 & & \text{where} \\
 & & \quad \mathit{sctc} = \mathtt{flat}(\mathtt{string?}) \mapsto \mathtt{flat}(\mathtt{void?})
\end{array}$$

The `mon` term monitors the module import that embeds the mixin into its use in the typed component with a contract ctc. The model encodes the untyped mixin from that example as a function from class c to an extension with method s—short for search. The contract ctc corresponds to the type in the example where sctc is the corresponding contract for s.

The monitored mixin is applied to the $\mathit{editor\%}$ class, meaning the reduction starts from this evaluation context:

$$\mathtt{send}(\mathtt{new}([\,]\ \mathit{editor\%}), s, \mathtt{"x"})$$

First the monitor itself reduces to a guarded function that installs a sealing contract on application. Next we reduce the application of the guarded mixin to the $\mathit{editor\%}$, sealing $\mathit{editor\%}$ as it is substituted in the mixin's body.

Here is the result of these first few reduction steps:

$$\begin{array}{l}
\mathtt{mon}_j^{u,t}(\mathtt{unseal/c}([s\ \mathit{sctc}]\,, \varsigma), \\
\qquad \mathtt{class/v}(\mathtt{SG}_j^{t,u}\{\mathit{editor\%}, [-]\,, \emptyset, \varsigma\}) \\
\qquad\quad \{\mathit{sc}\,x.\,\mathtt{send}(\mathtt{this},\ \mathtt{save\text{-}file},\ 0)\ -\})
\end{array}$$

The superclass of the class value is a sealed version of the $\mathit{editor\%}$ class; the unsealing contract remains.

The unsealing yields a guarded class:

$$\begin{array}{l}
\mathtt{G}_j^{u,t}\{\mathtt{class/v}(\mathtt{G}_j^{u,t}\{\mathit{editor\%}, [-]\})\{ \\
\qquad\quad \mathit{sc}\,x.\,\mathtt{send}(\mathtt{this},\ \mathtt{sf},\ 0)\}, \\
\quad [s\ \mathit{sctc}]\}
\end{array}$$

Finally, this guarded class is instantiated as an object containing the class. At this point, the method invocation of s is evaluated, which sets up the body of s as the redex. The body, in turn, calls the `sf` method. This method invocation

$\langle E[\texttt{blame}^k_j], S\rangle$	$\hookrightarrow$	$\langle \texttt{blame}^k_j, S\rangle$	
$\langle E[\cdots], S\rangle$	$\hookrightarrow$	$\langle E[\cdots], S\rangle$	
$\texttt{class}(cv)\,\{\overline{f := v}\ \ \overline{m^p x^p.\, e^p}\ \ \overline{m^o x^o.\, e^o}\}$	$.$	$\texttt{class/v}^\iota(cv)\,\{\overline{(f, v)}\ \ \overline{mx.e}\}$	CLASS
if $\overline{m^p} \cap \mathsf{Methods}(cv) = \emptyset$, $\overline{m^o} \subseteq \overline{\mathsf{Exposed}(cv)}$, $\overline{m^p} \subseteq \overline{\mathsf{NotSealed}(cv)}$			
and where ι is fresh and $\overline{mx.e} = \overline{m^p x^p.\, [\square/^{id}\iota]\, e^p} \oplus \overline{m^o x^o.\, [\square/^{id}\iota]\, e^o}$			
$\texttt{class}(cv)\,\{\overline{f := v}\ \ \overline{m^p x^p.\, e^p}\ \ \overline{m^o x^o.\, e^o}\}$	$\cdot$	$\texttt{blame}^l_j$	CLASSERR
if $\mathsf{HasBarrier}(cv, \overline{m})$, $\overline{m^p_i} \cap \mathsf{Methods}(cv) = \emptyset$, $\overline{m^o_i} \subseteq \mathsf{Methods}(cv)$			
and where $\overline{m} = \overline{m^p} \oplus \overline{m^o}$ and $(l, j) = \mathsf{LocateBarrier}(cv, \overline{m})$			
$\texttt{send}(o,\ m,\ v)$	$.$	$(\left[\texttt{this}/\, e_o, \overline{a/f}\right]\ e\ \ v)$	SEND
if $m \in \mathsf{Methods}(o)$ and $\mathsf{SameOwner}(o, m, \bot, \bot)$			
and where $e = \mathsf{Pull}(\lvert o\rvert^l, m)$, $e_o = \mathsf{ProtectThis}(\lvert o\rvert^l, m, \lvert o\rvert^l)$,			
and $\overline{(f, a, \iota)} = \mathsf{ObjectFields}(o)$			
$\texttt{send}(o,\ m,\ v)$	$.$	$\texttt{blame}^k_j$	SENDERR
if $m \in \mathsf{Methods}(o)$, and not $\mathsf{SameOwner}(o, m, \bot, \bot)$			
and where $(k, j) = \mathsf{OwnerLimit}(o, m, \bot, \bot, \bot)$			
$\texttt{mon}^{k,l}_j(\texttt{flat}(op),\ v)$	$\cdot$	$\texttt{if}\ op(v)\ v\ \texttt{blame}^k_j$	FLATC
$\texttt{mon}^{k,l}_j(c_1 \mapsto c_2,\ v)$	$\cdot$	$\lambda x.\texttt{mon}^{k,l}_j(c_2,\ (v\ \texttt{mon}^{l,k}_j(c_1,\ x)))$	FUNC
$\texttt{mon}^{k,l}_j(\texttt{class/c}^\bullet(\overline{[m\ c_1 \mapsto c_2]}),\ cv)$	$\cdot$	$\texttt{G}^{k,l}_j\{cv, \overline{[m\ c_1 \mapsto c_2]}\}$	CLASSC
if $\overline{m} = \mathsf{Exposed}(cv)$			
$\texttt{mon}^{k,l}_j(\texttt{class/c}^\bullet(\overline{[m\ c_1 \mapsto c_2]}),\ cv)$	$\cdot$	$\texttt{blame}^k_j$	CLASSCERR
if $\overline{m} \neq \mathsf{Exposed}(cv)$			
$\texttt{mon}^{k,l}_j(\texttt{object/c}(\overline{[m\ c_1 \mapsto c_2]}),\ o)$	$\cdot$	$\texttt{OG}^{k,l}_j\{o, \overline{[m\ c_1 \mapsto c_2]}\}$	OBJECTC
if $\overline{m_i} \subseteq \mathsf{Exposed}(o)$			
$\texttt{mon}^{k,l}_j(\texttt{object/c}(\overline{[m\ c_1 \mapsto c_2]}),\ o)$	$\cdot$	$\texttt{blame}^k_j$	OBJECTCERR
if $\overline{m} \not\subseteq \mathsf{Exposed}(o)$			
$\texttt{mon}^{k,l}_j(\texttt{seal/c}(\overline{[m\ c_1 \mapsto c_2]}, \overline{m'}, \varsigma),\ cv)$	$\cdot$	$\texttt{SG}^{l,k}_j\{cv, \overline{[m\ c_1 \mapsto c_2]}, \overline{m'}, \varsigma\}$	SEALC
if $\overline{m_i} \subseteq \mathsf{Exposed}(cv)$ and $\overline{m'}\backslash\overline{m} \cap \mathsf{Exposed}(cv) = \emptyset$			
$\texttt{mon}^{k,l}_j(\texttt{seal/c}(\overline{[m\ c_1 \mapsto c_2]}, \overline{m'}, \varsigma),\ cv)$	$\cdot$	$\texttt{blame}^k_j$	SEALCERR
if $\overline{m_i} \not\subseteq \mathsf{Exposed}(cv)$ or $\overline{m'}\backslash\overline{m} \subseteq \mathsf{Exposed}(cv)$			
$\texttt{mon}^{k,l}_j(\texttt{unseal/c}(\overline{[m\ c_1 \mapsto c_2]}, \varsigma)\rho\varsigma,\ cv)$	$\cdot$	$\texttt{G}^{k,l}_j\{\mathsf{Unseal}(cv, \varsigma), \overline{[m\ c_1 \mapsto c_2]}\}$	UNSEALC
if $\mathsf{Sealed}(cv, \varsigma)$ and $\overline{m_i} \subseteq \mathsf{Exposed}(cv)$			
$\texttt{mon}^{k,l}_j(\texttt{unseal/c}(\overline{[m\ c_1 \mapsto c_2]}, \varsigma)\rho\varsigma,\ cv)$	$\cdot$	$\texttt{blame}^k_j$	UNSEALCERR
if not $\mathsf{Sealed}(cv, \varsigma)$ or $\overline{m_i} \not\subseteq \mathsf{Exposed}(cv)$			
$\texttt{mon}^{k,l}_j(\forall^{\mathbf{c}}(\rho\backslash\overline{m}).(c_1 \mapsto c_2),\ v)$	$\cdot$	$\forall\texttt{G}^{k,l}_j(\rho\backslash\overline{m}).(c_1 \mapsto c_2)\{v\}$	$\forall$C
$(e\ \ v)$	$\cdot$	$(\texttt{mon}^{k,l}_j(c_1 \mapsto c_2[\![\rho/\varsigma]\!],\ v_1)\ v)$	$\forall$G-APP
where $e = \forall\texttt{G}^{k,l}_j(\rho\backslash\overline{m}).(c_1 \mapsto c_2)\{v_1\}$ and ς is fresh.			

Figure 12. Reductions for monitored terms

name	domain, range / purpose
HasBarrier	o or cv, $\overline{m} \mapsto$ `#t` or `#f` checks if some m is inaccessible due to a guard.
LocateBarrier	o or cv, $\overline{m} \mapsto (l, l)$ returns the blame labels for the closest inaccessible m in the object or class hierarchy.
SameOwner	o or cv, l or $\perp$, l or $\perp \mapsto$ `#t` or `#f` checks if m is owned by the calling context
OwnerLimit	o or cv, l or $\perp$, l or $\perp$, l or $\perp \mapsto (l, l)$ returns the label of the calling context of m and of the contract boundary where ownership of m was lost
Pull	$cv, m \mapsto e$ returns m's implementation as a λ-term wrapped with the necessary contracts
ProtectThis	$o, m, o \mapsto o$ traverses the first object and its hierarchy to apply all contracts needed to protect the second object when it is a receiver of a call to m
Sealed	$cv, \varsigma \mapsto$ `#t` or `#f` checks if cv contains a seal guard that is locked with ς.
Unseal	$cv, \varsigma \mapsto cv$ removes seal guards locked with ς in the class hierarchy.
Notsealed	o or $cv \mapsto \overline{m}$ returns unsealed methods in the hierarchy.
Exposed	$cv \mapsto \overline{m}$ returns all the exposed method names in the hierarchy
Methods	o or $cv \mapsto \overline{m}$ returns method names in the hierarchy.
Fields	$cv \mapsto \overline{(f, v, \iota)}$ returns the fields' initial values in the class hierarchy.
ObjectFields	$o \mapsto \overline{(f, a, \iota)}$ returns the object's field values

Figure 13. Metafunctions are defined inductively on the structure of the first argument

triggers the SameOwner metafunction with the `sf` method as an argument. This metafunction fails because the method lacks a contract between the method's invocation on `this` and the implementation in the *editor%* class in the hierarchy. Thus, the final result is $\texttt{blame}^{u}_{j}$ as expected.

4. Type Soundness for Mixed Programs

Type soundness establishes a minimal logical standard for a programming language. In this section, we use the formal model to prove that our design meets this criterion. Our proof of soundness requires two steps. First, we must prove that the type system is sound with respect to the execution of typed programs. Second, we must show that mixing in untyped components does not violate the invariants of the typed components. That is, the interpretation of types as contracts at component boundaries must prevent any type-like runtime error assignable to typed components.

For the proof of soundness, we employ the usual progress and preservation technique [37]. For the soundness of types as contracts, we show that the contract system is a *complete monitor* [6], meaning components do not export their values without appropriate contract protection. Based on these two major steps, we finally show that typed components of mixed-type programs cannot be blamed for violations of type invariants.

4.1 Type soundness

The reduction relation we use for typed terms is the relation for untyped terms in figure 12 except that the reductions carry along type annotations. Type applications are reduced analogous to function application, but types do not affect reductions in any other way.

In this setting, typed programs cannot go wrong, in particular they cannot call undefined methods.

Lemma 1. *(Type Soundness) For all e_τ, if $\emptyset \,|\, \emptyset \vdash e_\tau : \tau$, either*

- *for all e_1 such that $\langle e_\tau, \emptyset\rangle \hookrightarrow^* \langle e_1, S_1\rangle$, there exists an e_2 such that $\langle e_1, S_1\rangle \hookrightarrow \langle e_2, S_2\rangle$ or,*
- *$\langle e_\tau, \emptyset\rangle \hookrightarrow^* \langle v_\tau, S\rangle$ where $\emptyset \,|\, \Sigma \vdash v_\tau : \tau'$ for some Σ and $\emptyset \,|\, \Sigma' \vdash S'$ and $\tau' <: \tau$.*

$\mathcal{G}; l \Vdash e$	Well formed programs
$\mathcal{D}; \overline{k}; \overline{l} \rhd c$	Well formed source contracts
$\mathcal{K}; \mathcal{G}; \mathcal{S}; l \Vdash e$	Well formed terms
$\mathcal{K}^{k,l}; \mathcal{D}; \overline{k}; \overline{l} \rhd c$	Well formed contracts
$\mathcal{K} \Vdash \mathcal{S} \sim S$	Well formed stores

Figure 14. Relations for Well Formed Programs

Recall from section 3 that $\Gamma \mid \Sigma \vdash S$ says store S is typable under Γ and Σ. As mentioned, the theorem requires two conventional lemmas: progress and preservation. The statement of both calls for a typing judgment applicable to intermediate states, i.e., states with non-empty stores for private fields. Other than that, the details are straightforward and omitted.

4.2 Complete monitoring

Complete monitoring [6] is a formal criterion for the correct design of a contract system. It imposes two conditions on a correct contract system: *complete mediation* and *correct blame assignment*.

Complete mediation requires that the contract system does not allow values to pass between components without a contract check—possibly the always-true contracts—where contract monitors act as component boundaries. Put differently, at any point in time every value is owned by one and only one component. Intuitively, *owner* denotes the component that may affect the flow of the value (e.g., export the value to another component). If the owner wishes to share a value with other components, it must do so under the auspices of a monitor or guard. Ownership of terms is formalized via *ownership labels*. Expressions *e* are annotated with an owner label l, e.g., $|\, e\,|^{l}$. We use l_o to indicate the implicit owner of the whole program.

Correct blame assignment requires that upon contract failure, the contract system blames the contract party responsible for the breach of the contract. A party is responsible for a contract failure if a value it owns fails an immediate contract check and it is obliged to uphold that particular contract. To indicate responsibility for immediate contract checks, we use obligation annotations. For instance, $\texttt{flat}(op)$ is annotated with obligation l as $\lfloor \texttt{flat}(op) \rfloor^{l}$. If a component l has an immediate contract as an obligation, it must ensure that when a value crosses the boundary protected by this contract, the contract is checked appropriately.

Complete monitoring assumes that component boundaries agree with ownership annotations and that obligations match the labels on monitors. To check this relationship, we define a relation $\mathcal{G}; l \Vdash e$ that ensures the well-formedness of e with respect to a label l and environment $\mathcal{G}$. According to the relation's judgments, ownership can change only via monitors. When this happens, the ownership annotation on the guarded expression must match the positive label of the monitor while the negative label should coincide with the owner of the context. The judgment does not allow any other ownership annotations that may change the owner of a term. The environment $\mathcal{G}$ records the owner of each binding encountered by the judgments.

The relation for well-formed programs relies on the definition of an additional judgment $\mathcal{D}; \overline{k}; \overline{l} \rhd c$ that checks that contracts are well-formed. It picks up the positive and negative labels of each monitor and ensures they coincide with the obligation annotations on the positive and negative pieces of the contract of the monitor. After all, the client (negative label) of a guarded component is responsible for the values it consumes while the server (positive label) is responsible for the values it produces.

Unfortunately, this strict distinction between negative and positive parties does not hold for classes and methods. The receiver of a method invocation is an object that, via substitution, traverses the class hierarchy from the call site to the method implementation. Depending on the direction of this migration, the roles of clients and servers on the contract boundaries reverse. Thus, for each method contract the server and the client share responsibility for all pieces of the contract. We account for this codependence by allowing obligation annotations to have sets of labels and by adjusting the judgments accordingly.

Sealing and unsealing contracts pose further challenges for proving the correctness of the contract system. We guarantee that sealing contracts show up only in negative positions with respect to the corresponding row variable ρ by marking sealing contracts with the annotation, $\rho = -$. Similar annotations, $\rho = +$, decorate unsealing contracts. The environment $\mathcal{D}$ tracks the bound row variables and their polarity, $+$ or $-$. The well-formed source contracts relation ensures that sealing contracts appear in the right places.

The judgments for well-formed programs and contracts provide the foundation for complete monitoring. Complete monitoring aims to establish that if each term in a program has a single owner initially, then each term has a single owner throughout the evaluation of the program. Embedded *foreign* terms can exist inside a *host* component, but only if they are wrapped with a contract monitor. Accordingly, we change the reduction semantics to propagate ownership annotations as values flow through the program. This change is independent of the contract system and does not affect the meaning of the program. Host components assimilate values only when the values are primitive and satisfy their contracts. In any other case, values accumulate more owners as they flow from one component to another.

If a contract system is a complete monitor, then all annotations in a value's stack of ownership labels must be identical. Otherwise, the contract system would allow values to cross component boundaries without appropriate protection. We make redexes that involve values with stacks of non-identical owners stuck states, which makes violations of the single-owner policy manifest. To differentiate between the latter stuck states and stuck states due to type errors we introduce $\texttt{error}^k$, dynamic type errors. Thus, a contract system that is a complete monitor renders states unreachable if they are stuck due to a value having multiple owners.

Definition 1. *(Complete monitoring) A reduction relation $\rightarrow$ for TFCC is a complete monitor if $\emptyset; l_0 \Vdash e_0$ implies:*

- $\langle e_0, \emptyset \rangle \rightarrow^* \langle v, S \rangle$ *or,*
- $\langle e_0, \emptyset \rangle \rightarrow^* \langle \texttt{error}^k, S \rangle$ *or,*
- *for all e_1 such that $\langle e_0, \emptyset \rangle \rightarrow^* \langle e_1, S_1 \rangle$, there exists e_2 such that $\langle e_1, S_1 \rangle \rightarrow \langle e_2, S_2 \rangle$ or,*
- *if $\langle e_0, \emptyset \rangle \rightarrow^* \langle e_1, S_1 \rangle \rightarrow^* \langle \texttt{blame}^k_j, S_2 \rangle$, there exists*
  - $e_1 = \texttt{mon}^{k,l}_j(\lfloor \texttt{flat}(op) \rfloor^{\overline{k}}, v)$ *and for all such e_1, $v = |v_1|^k$ and $k \in \overline{k}$,*
  - $e_1 = \texttt{mon}^{k,l}_j(\lfloor \texttt{class/c}^{\bullet}(\overline{[m\ c_1 \mapsto c_2]}) \rfloor^{\overline{k}}, v)$ *and for all such e_1, $v = |v_1|^k$ and $k \in \overline{k}$ or,*
  - $e_1 = \texttt{mon}^{k,l}_j(\lfloor \texttt{object/c}(\overline{[m\ c_1 \mapsto c_2]}) \rfloor^{\overline{k}}, v)$ *and for all such e_1, $v = |v_1|^k$ and $k \in \overline{k}$ or,*
  - $e_1 = \texttt{mon}^{k,l}_j(\lfloor \overset{\rho=+}{\texttt{unseal/c}}(\overline{[m\ c_1 \mapsto c_2]}, \varsigma) \rfloor^{\overline{k}}, v)$ *and for all such e_1, $v = |cv|^k$ and $k \in \overline{k}$ or,*
  - $e_1 = \texttt{mon}^{k,l}_j(\lfloor \overset{\rho=-}{\texttt{seal/c}}(\overline{m'}, \overline{[m\ c_1 \mapsto c_2]}, \varsigma) \rfloor^{\overline{k}}, v)$ *and for all such e_1, $v = |cv|^k$ and $k \in \overline{k}$ or,*
  - $e_1 = \texttt{send}(o, m, v)$ *where* $\mathsf{OwnerLimit}(o, m, \bot, \bot, \bot) = (k, j)$ *or,*
  - $e_1 = \texttt{class}(cv)\ \{\overline{f_i := v_i} \quad \overline{m^p x^p.\, e^p} \quad \overline{m^o x^o.\, e^o}\}$ *and where* $\mathsf{LocateBarrier}(cv, \overline{m}) = (k, j)$*, and* $\overline{m} = \overline{m^p} \oplus \overline{m^o}$.

In addition to eliminating stuck states, the definition of complete monitoring requires that the system blames a component only when a violation is due to one of the component's unmet obligations. Thus, our definition includes cases for all immediate checks that the contract system makes and, furthermore, includes cases that ensure sealing and unsealing are handled properly.

Lemma 2. $\hookrightarrow$ *is a complete monitor for TFCC.*

To prove that $\hookrightarrow$ is a complete monitor for TFCC we employ subject reduction. The proof combines a progress and a preservation lemma which establish that reduction preserves the subject and does not lead to stuck states:

Progress. *If $\mathcal{K}; \mathcal{G}; \mathcal{S}; l_0 \Vdash e_0$ and $\mathcal{K} \Vdash \mathcal{S} \sim S_0$ then $\langle e_0, S_0 \rangle \hookrightarrow \langle e_1, S_1 \rangle$ or $e_0 = v$ or $e_0 = \texttt{error}^k$ or $e_0 = \texttt{blame}^k_j$.*

Preservation. *If $\mathcal{K}; \emptyset; \mathcal{S}; l_0 \Vdash e_0$, $\mathcal{K} \Vdash \mathcal{S} \sim S_0$ and $\langle e_0, S_0 \rangle \hookrightarrow \langle e_1, S_1 \rangle$ then there exist $\mathcal{K}'$,$\mathcal{S}'$ such that $\mathcal{K} \subseteq \mathcal{K}'$, $\mathcal{S} \subseteq \mathcal{S}'$, $\mathcal{K}'; \emptyset; \mathcal{S}'; l_0 \Vdash e_1$ and $\mathcal{K}' \Vdash \mathcal{S}' \sim S_1$.*

The two lemmas use a subject different than the judgment for well-formed programs. Even though the latter guarantees the absence of stuck states and correct blame, it is not general enough to handle terms produced during evaluation. The judgment for well-formed terms, $\mathcal{K}; \mathcal{G}; \mathcal{S}; l \Vdash e$, is an extension of the relation for well-formed programs that can accommodate intermediate terms such as objects, class, values and guards.

Proposition 1. *If $\emptyset; l_0 \Vdash e_0$ then $\emptyset; \emptyset; \emptyset; l_0 \Vdash e_0$.*

The new relation uses two additional environments, $\mathcal{S}$ and $\mathcal{K}$. The first, *store coloring*, records the owner of the contents of each store location. Its function is similar to that of store typing in type soundness proofs. The relation $\mathcal{K} \Vdash \mathcal{S} \sim S$ ensures that store coloring is synchronized with the contents of the store.

The second, *key coloring*, records the owner of each sealing/unsealing key. Components obtain keys as they generate them and each component should only seal and unseal values with keys it owns. The environment $\mathcal{K}$ guarantees that the keys stored in sealing guards have the appropriate owner.

In addition, key coloring checks whether contracts remain well-formed after row variables are replaced by keys. We specify this constraint with an extension of the relation for well-formed contracts, $\mathcal{K}^{k,l}; \mathcal{D}; \overline{k}; \overline{l} \rhd c$. It refers to the key coloring when it checks sealing and unsealing contracts. In these cases, the owner of the seal must match the party that performs the sealing or the unsealing. We determine this party by picking up the labels from monitors, decorate key coloring with them, and swapping their position when we go from positive to negative position as we traverse contracts. If a sealing contract is in a positive position, the owner of the key should be the server of the monitor since it is the component that must protect the sealed value from its context. Similar constraints apply to the other cases for sealing and unsealing contracts.

4.3 The Blame Theorem

Now that we have established complete monitoring for TFCC, we can prove type soundness for programs that mix typed and untyped code.

From type soundness for typed code, we know that purely typed components are safe from type errors. In mixed programs, typed and untyped components interact by passing values through contract monitors. A monitor $\texttt{mon}^{u,t}_j(c,\, e)$ embeds an untyped value into a typed context because u is the server and t is the client. Similarly, a monitor $\texttt{mon}^{t,u}_j(c,\, e_\tau)$ embeds a typed value into an untyped context. Since the boundaries between untyped and typed components are monitored by the contract system, we need the contract system to protect values that flow across the boundaries as strongly as the type system. For that reason we impose contracts on the interfaces between typed and untyped components to simulate the types that the typed components expect. Of course, untyped components may not live up to

$\mathrm{K} \,\vert\Gamma\, \vert\Sigma \vdash e : \tau$	Well typed mixed terms
$\mathrm{K} \,\vert\Gamma\, \vert\Sigma \vdash e$	Well formed mixed terms
$\mathrm{K} \,\vert\Gamma\, \vert\Sigma \vdash_u cv \sim r$	Well typed mixed class values
$\mathrm{K} \,\vert\Gamma\, \vert\Sigma \vdash_u o \sim r$	Well typed mixed objects

Figure 15. Relations for Well Typed Mixed Programs

the expectations that types express, but the contracts catch such impedance mismatches.

In a sound system, since typed code always respects the type discipline, a typed component should never break any contract used to mediate values between typed and untyped code. Intuitively, soundness for mixed programs means that the contract system never blames a typed component.

The first step to formalize soundness is to extend the notion of well-typed terms to include mixed programs. The first two relations in figure 15 jointly express these roles. The first one type-checks typed components using the contracts-to-types correspondence to create a type for embedded untyped components. The second relation traverses untyped components and delegates type-checking of embedded typed components to the first relation. Again we use the contract on the interface between an untyped and a typed component and the contracts to types correspondence from figure 9 to construct a type to check against the typed code.

Blame Theorem. *If* $\emptyset \,\vert\emptyset\, \vert\emptyset \vdash e$*, then* $\langle e, \emptyset\rangle \not\hookrightarrow^* \langle \mathtt{error}^t, S\rangle$ *and* $\langle e, \emptyset\rangle \not\hookrightarrow^* \langle \mathtt{blame}^t_j, S\rangle$.

The cornerstone for proving the blame theorem is a preservation lemma for mixed terms. According to complete monitoring the only source of blame for typed components is a violation of a first-order contract check on a typed value. Due to type soundness for typed code, however, if all first-order contracts that guard typed values correspond to the type of the values, the checks never fail. Indeed, the preservation lemma for mixed programs guarantees that reduction always results in well-typed mixed terms. This implies that all first-order checks have the desired property.

Lemma 3. *(Mixed Preservation) If* $\mathrm{K}_0 \,\vert\emptyset\, \vert\Sigma_0 \vdash e_0$, $\emptyset \,\vert\, \Sigma_0 \vdash S_0$ *and* $\langle e_0, S_0\rangle \hookrightarrow \langle e_1, S_1\rangle$ *then there exist* K_1,Σ_1 *such that* $\mathcal{K}_0 \subseteq \mathrm{K}_1$, $\Sigma_0 \subseteq \Sigma_1$, $\mathrm{K}_0 \,\vert\emptyset\, \vert\Sigma_1 \vdash e_1$ *and* $\emptyset \,\vert\, \Sigma_1 \vdash S_1$.

The proof of mixed preservation demands an extension of the top two relations of figure 15 to intermediate terms. Sealing and unsealing contracts make this extension challenging. As explained, in section 3, sealing is a mechanism for hiding details of a class provided to a component. Unsealing makes the hidden details available again to the owner of the class after the component returns its result. Thus when we seal a typed class the contracts on the sealing interface do not fully reflect the type of the sealed class but only its visible parts. We overcome this obstacle and prove preservation even in the presence of sealing using the K environment, which maps keys to row types and we update it upon mixin application. For uses of untyped mixins from typed code, we map the newly created key to the row provided with row application. In the reverse case we map the new key to the empty row.

Recall that after unsealing a class value that contains a sealed superclass, the sealed methods become reachable again. However the contracts in the class hierarchy do not reflect the types of the now available methods. The last two relations of figure 15 reconstruct these types via a traversal of the class hierarchy that propagates the types of typed methods appropriately.

Constructing the types of untyped code is not the only difficulty of proving preservation. In fact, it is even more crucial to establish that typed and untyped code do not intermingle in an unprotected manner. Fortunately, complete monitoring solves this problem and guarantees that a monitor or a guard always mediates the interaction between a typed and an untyped component. This insight significantly reduces the effort of proving preservation as we just have to ensure that the contract we attach to a migrating typed value corresponds to the value's type. Put differently, the proof of the blame theorem is reduced to showing that monitors and guards related reductions of well-typed mixed terms result in well-typed mixed terms.

5. Related Work

The body of related work consists of research on combining typed and untyped languages, polymorphic contracts, and types for extensible objects or records.

Gradual typing Gray et al. [16] present a multi-language system that integrates Scheme and Java, using contracts and mirrors to mediate interactions between the two languages. Their system does not handle dynamic class composition because class values cannot flow from Scheme to Java. In subsequent work, Gray [14] models interoperation between Java and Scheme with cross-language inheritance, but does not address dynamic class composition mechanisms such as mixins or first-class classes. Closer to our work, Gray [15] also models interoperation between Java and JavaScript, allowing JavaScript code to use Java classes as prototypes. While her model describes dynamic class composition, stringent restrictions are placed on inheritance to make composition safe. In particular, Java classes cannot dispatch to Javascript extensions.

Siek and Taha [28] use an object calculus for the formulation of their gradual typing system for OO programming. Their system handles subtyping for objects, but since their language contains neither classes nor extensible objects, there is no treatment of inheritance between untyped and typed code.

Another approach to combining untyped and typed code is applying type reconstruction to an untyped language. DRuby [12] is such a system for a subset of Ruby. While Ruby supports dynamic class composition via a mixin feature called *modules*, DRuby's type system is unable to support the runtime composition of classes that Ruby allows.

Bierman et al. [2] formalize the `dynamic` type that is available in C$^\sharp$4.0. The latter's dynamic types make no provision for higher-order data, however.

Runtime sealing Morris [24] proposed sealing in 1973 as a linguistic mechanism to prevent access to private pieces of code. More recently, sealing has been used to protect datatype abstraction [25].

Guha et al. [17] investigate parametricity guarantees in untyped languages via a combination of contracts and sealing They use so-called coffers—an opaque datatype for sealing—to wrap arguments to contracted parametric functions. Matthews and Ahmed [21] provide a formal validation of this approach in a multi-language system combining the untyped λ-calculus and System F, in which sealing is applied at language boundaries to ensure parametricity. Ahmed et al. [1] present a finer-grained sealing mechanism that associates seals with type abstractions. Our approach extends this line of work by adding partial seals that allow code to see some aspects of classes rather than blocking out all class features. Notice, however, that our proof technique for type soundness differs significantly. Instead of the logical relation of Matthews and Ahmed or the subtyping-based simulation of Ahmed et al., we base our proof on complete monitoring. This enables a modular proof structure and the use of standard subject reduction for establishing type soundness.

Types for dynamic class composition Row polymorphism originated as a mechanism to enable type inference for objects. Wand [36] proposes a type inference algorithm for a simple object-oriented language based on recursive records. These ideas were also adopted into type systems for extensible records [4, 13, 18].

Closely related is ML-ART [26], which is capable of encoding first-class classes, but does not provide classes as a primitive concept. ML-ART does not allow subsumption on objects, thus disallowing the familiar polymorphism over objects. In the same vein, Fisher [10] presents a calculus with typed extensible objects using row polymorphism. Her calculus also enables an encoding of class-based programming, but does not support them as primitive features. Bono et al. [3] build on this line of work, and include classes and mixins as primitives in their calculus. Their calculus covers the use cases of mixins, but their classes do not support other dynamic uses of first-class classes. None of these models were designed for a gradually-typed setting.

6. Status and Outlook

This paper presents the first design for a gradual typing system that accommodates class composition across components with distinct type disciplines. On the typed side, a novel use of row polymorphism allows for specifying interface types for mixins, traits, and other manipulations of first-class classes. On the untyped side, the introduction of sealing contracts allows the system to seal off portions of the specialization interface of classes; this sealing ensures the safety of first-class classes as they flow from the typed world to the untyped one and back. Finally, the paper also includes a new, easy-to-use proof technique for the proof of blame theorems in the presence of polymorphism.

Our work puts us in a position from where we can explore the pragmatics of gradual typing for first-class classes. In the near future, we intend to implement this type system and use it to equip a significant portion of our code base with types.

Acknowledgments Christos Dimoulas and Sam Tobin-Hochstadt thank Amal Ahmed for several discussions about polymorphic contracts and blame. The authors are also grateful to Stephen Chang, Vincent St-Amour, Aaron Turon, and Arjun Guha for comments on early drafts.

References

[1] AHMED, A., FINDLER, R. B., SIEK, J. G., AND WADLER, P. Blame for all. In *Symposium on Principles of Programming Languages* (2011), pp. 201–214.

[2] BIERMAN, G., MEIJER, E., AND TORGERSEN, M. Adding dynamic types to C$^\sharp$. In *European Conference on Object-Oriented Programming* (2010), pp. 76–100.

[3] BONO, V., PATEL, A., AND SCHMATIKOV, V. A core calculus of classes and mixins. In *European Conference on Object-Oriented Programming* (1999), pp. 43–66.

[4] CARDELLI, L., AND MITCHELL, J. C. Operations on records. *Mathematical Structures in Computer Science 1* (1991), 3–48.

[5] DIMOULAS, C., FINDLER, R. B., FLANAGAN, C., AND FELLEISEN, M. Correct blame for contracts: No more scapegoating. In *Symposium on Principles of Programming Languages* (2011), pp. 215 – 226.

[6] DIMOULAS, C., TOBIN-HOCHSTADT, S., AND FELLEISEN, M. Complete monitoring for behavioral contracts. In *European Symposium on Programming* (2012), pp. 211–230.

[7] FELLEISEN, M., FINDLER, R. B., AND FLATT, M. *Semantics Engineering with PLT Redex.* MIT Press, 2009.

[8] FINDLER, R. B., CLEMENTS, J., FLANAGAN, C., FLATT, M., KRISHNAMURTHI, S., STECKLER, P., AND FELLEISEN, M. DrScheme: A programming environment for Scheme. *Journal of Functional Programming 12*, 2 (2002), 159–182.

[9] FINDLER, R. B., AND FELLEISEN, M. Contracts for higher-order functions. In *International Conference on Functional Programming* (2002), pp. 48–59.

[10] FISHER, K. S. *Type Systems for Object-Oriented Programming Languages.* PhD thesis, Stanford University, 1996.

[11] FLATT, M., FINDLER, R. B., AND FELLEISEN, M. Scheme with classes, mixins, and traits. In *Asian Symposium on Programming Languages and Systems* (2006), pp. 270–289.

[12] FURR, M., AN, J.-H. D., FOSTER, J. S., AND HICKS, M. Static type inference for Ruby. In *Symposium on Applied Computing* (2009), pp. 1859–1866.

[13] GASTER, B. R., AND JONES, M. P. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, University of Nottingham, 1996.

[14] GRAY, K. E. Safe cross-language inheritance. In *European Conference on Object-Oriented Programming* (2008), pp. 52–75.

[15] GRAY, K. E. Interoperability in a scripted world: Putting inheritance & prototypes together. In *Foundations of Object-Oriented Languages* (2011).

[16] GRAY, K. E., FINDLER, R. B., AND FLATT, M. Fine-grained interoperability through contracts and mirrors. In *Object-Oriented Programming, Systems, Languages, and Applications* (2005), pp. 231–245.

[17] GUHA, A., MATTHEWS, J., FINDLER, R. B., AND KRISHNAMURTHI, S. Relationally-parametric polymorphic contracts. In *Dynamic Languages Symposium* (2007), pp. 29–40.

[18] HARPER, R., AND PIERCE, B. A records calculus based on symmetric concatenation. In *Symposium on Principles of Programming Languages* (1991), pp. 131–142.

[19] HERMAN, D., AND FLANAGAN, C. Status report: Specifying JavaScript with ML. In *ML Workshop* (2007).

[20] LAMPING, J. Typing the specialization interface. In *Object-Oriented Programming, Systems, Languages, and Applications* (1993), pp. 201–214.

[21] MATTHEWS, J., AND AHMED, A. Parametric polymorphism through run-time sealing, or, theorems for low, low prices! In *European Symposium on Programming* (2008), pp. 16–31.

[22] MATTHEWS, J., AND FINDLER, R. B. The meaning of multi-language programs. In *Symposium on Principles of Programming Languages* (2007), pp. 3–10.

[23] MEJER, E., AND DRAYTON, P. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA Workshop on Revival of Dynamic Languages* (2004).

[24] MORRIS, JR., J. H. Types are not sets. In *Symposium on Principles of Programming Languages* (1973), pp. 120–124.

[25] PIERCE, B., AND SUMII, E. A bisimulation for dynamic sealing. In *Symposium on Principles of Programming Languages* (2004), pp. 161–172.

[26] RÉMY, D. Programming objects with ML-ART an extension to ML with abstract and record types. In *Theoretical Aspects of Computer Software* (1994), pp. 321–346.

[27] SIEK, J., AND TAHA, W. Gradual typing for functional languages. In *Workshop on Scheme and Functional Programming* (2006), pp. 81–92.

[28] SIEK, J., AND TAHA, W. Gradual typing for objects. In *European Conference on Object-Oriented Programming* (2007), pp. 2–27.

[29] STRICKLAND, T. S., AND FELLEISEN, M. Contracts for first-class classes. In *Dynamic Languages Symposium* (2010), pp. 97–111.

[30] TAKIKAWA, A., STRICKLAND, T. S., DIMOULAS, C., TOBIN-HOCHSTADT, S., AND FELLEISEN, M. Gradual typing for first-class classes. Technical Report NU-CCIS-12-02, Northeastern University, College of Computer and Information Science, 2012.

[31] TANG, A. Perl 6: reconciling the irreconcilable. Symposium on Principles of Programming Languages, 2007.

[32] TOBIN-HOCHSTADT, S., AND FELLEISEN, M. Interlanguage migration: from scripts to programs. In *Dynamic Languages Symposium* (2006), pp. 964–974.

[33] TOBIN-HOCHSTADT, S., AND FELLEISEN, M. The design and implementation of Typed Scheme. In *Symposium on Principles of Programming Languages* (2008), pp. 395–406.

[34] TOBIN-HOCHSTADT, S., AND FELLEISEN, M. Logical types for untyped languages. In *International Conference on Functional Programming* (2010), pp. 117–128.

[35] WAND, M. Complete type inference for simple objects. In *Symposium on Logic in Computer Science* (1987).

[36] WAND, M. Type inference for objects with instance variables and inheritance. In *Theoretical Aspects of Object-Oriented Programming* (1994), pp. 97–120.

[37] WRIGHT, A. K., AND FELLEISEN, M. A syntactic approach to type soundness. *Information and Control* (1994), 38–94.

[38] WRIGSTAD, T., NARDELLI, F. Z., LEBRESNE, S., ÖSTLUND, J., AND VITEK, J. Integrating typed and untyped code in a scripting language. In *Symposium on Principles of Programming Languages* (2010), pp. 377–388.