A Theory of Tagged Objects

Joseph Lee[1], Jonathan Aldrich[1], Troy Shaw[2], and Alex Potanin[2]

1 **Carnegie Mellon University**
Pittsburgh, PA, USA
josephle@andrew.cmu.edu, aldrich@cs.cmu.edu
2 **Victoria University of Wellington**
New Zealand
troyshw@gmail.com, alex@ecs.vuw.ac.nz

Abstract

Foundational models of object-oriented constructs typically model objects as records with a structural type. However, many object-oriented languages are class-based; statically-typed formal models of these languages tend to sacrifice the foundational nature of the record-based models, and in addition cannot express dynamic class loading or creation. In this paper, we explore how to model statically-typed object-oriented languages that support dynamic class creation using foundational constructs of type theory. We start with an extensible tag construct motivated by type theory, and adapt it to support static reasoning about class hierarchy and the tags supported by each object. The result is a model that better explains the relationship between object-oriented and functional programming paradigms, suggests a useful enhancement to functional programming languages, and paves the way for more expressive statically typed object-oriented languages. In that vein, we describe the design and implementation of the Wyvern language, which leverages our theory.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases objects, classes, tags, nominal and structural types

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.174

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.1.1.3

1 Introduction

Many models of statically typed object-oriented (OO) programming encode objects as records, usually wrapped inside a recursive or existential type [5]. These models elegantly capture many essential aspects of objects, including object methods, fields, dispatch, and subtyping. They are also foundational in that they describe core object-oriented constructs in terms of the fundamental building blocks of type theory, such as functions, records, and recursive types.

These models have been used to investigate a number of interesting features of OO languages. For example, dynamically-typed languages often support very flexible ways of constructing new objects and classes at run time, using ideas like mixins [10]. Typed, record-based models of object-oriented programming can support these flexible composition mechanisms with small, natural extensions [27].

Unfortunately, there are also important aspects of common object-oriented programming languages that are not adequately modeled by record-based object encodings. Many object-oriented languages are class-based: each object is an instance of a class, and classes are

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 174–197

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


related by a subclass (or inheritance) relationship. In statically-typed class-based languages, subtyping is generally not structural, but instead follows the subclassing relation. Most class-based languages also provide a way for programmers to test whether an object is an instance of a class, for example via a cast, which results in an error if the test fails, or an `instanceof` operation, which returns a boolean.

Typical models of these languages, such as Featherweight Java [18], assume that each object is a member of a class, and that a fixed class table exists mapping classes to their definitions (and thus defining a subclassing relation). Casts or `instanceof` tests can then be encoded using class table lookups. Such dynamic instance checks are useful in cases where a programmer wishes to express the behavior of a sum type in a statically-typed OO language, as in abstract syntax tree evaluation. Scala makes the relationship between sum-types and instance checking explicit through its pattern-matching expression, which can pattern match an object against multiple possible subclasses [8].

The class-based models of object-oriented programming with which we are familiar have two limitations. The first limitation is the fixed nature of the class table: the models assume that the entire class table can be computed at compile time, meaning that run-time loading, generation, or composition of classes is not modeled. These models, therefore, do not capture the dynamic compositions that are possible in class-based dynamic object-oriented languages such as Smalltalk. As a result, they are an unsuitable foundation for important areas of research, such as exploring future statically-typed object-oriented languages that support more dynamic forms of composition.

The second limitation is that these class-based models are not foundational in nature, in that they do not explain classes and instance checking in terms of the fundamental constructs of type theory. The more ad-hoc nature of these models inhibits a full understanding of the nature of classes and how classes relate to ideas in other paradigms, such as functional programming. For example, as described above, Scala's support for case classes that mirror datatypes in functional programming suggests an analogy between OO class types and type theory sums, and between instance checking in object-oriented languages and pattern matching or tag checking in functional languages. In particular, the Standard ML language provides extensible sums through its `exn` type. This poses a natural question: could an `exn`-like construct, suitably extended, be used to model `instanceof` checks in object-oriented languages?

The primary contribution of this paper is exploring a more flexible and foundational model of class-based object-oriented programming languages. At a more detailed level, the contributions include:

- Section 2 defines a source-level statically-typed object-oriented language that supports dynamic creation of class hierarchies, including mixin composition, along with a match construct that generalizes instanceof checks and casts. We also show examples demonstrating the expressive power of the language, including a technique for enforcing encapsulation, the memento design pattern, and type-safe dynamic mixin composition.
- Section 3 defines a foundational calculus, consisting of a standard core type theory with the addition of constructs to support hierarchical tag creation, tag matching, and the extraction of tagged content. We define the static and dynamic semantics of the foundational language and prove soundness.
- In Section 4, the semantics of the source-level language is defined by a translation into the target language, thus explaining how dynamic, class-based OO languages can be modeled in terms of type theory. The translation motivates the particular tag manipulation constructs of the foundational calculus, explaining how constructs such as the `exn` type in

$$e ::= \texttt{class}\ x\ \{\overline{f:\tau}, \overline{m:\tau=e}\}\ \texttt{in}\ e \mid \texttt{class}\ x\ \texttt{extends}\ x\ \{\overline{f:\tau}, \overline{m:\tau=e}\}\ \texttt{in}\ e$$
$$\mid \texttt{new}(x;\overline{e}) \mid e.f \mid e.m \mid \texttt{match}(e_1;x_1;x_2.e_2;e_3) \mid \lambda x{:}\tau.e \mid e\ e \mid \texttt{let}\ x=e\ \texttt{in}\ e$$
$$\mid \texttt{letrec}\ x=e\ \texttt{in}\ e \mid x$$

$$\tau ::= \texttt{class}\ x\ \{\overline{f:\tau}, \overline{m:\tau}\} \mid \texttt{class}\ x\ \texttt{extends}\ x\ \{\overline{f:\tau}, \overline{m:\tau}\} \mid \tau \to \tau \mid x\ \texttt{obj}$$

Figure 1 Source Language Syntax.

Standard ML need to be adapted to serve as a foundation for object-oriented programming.
- In Section 5, we present the publicly available implementation of our language.

After presenting these contributions, our paper discusses extensions in Section 6 and additional related work in Section 7. Section 8 concludes with a discussion of the potential applications of the model.

2 First-Class Classes

We motivate our work with a source language that treats classes as first-class expressions, putting them in the same syntactic category as objects, functions, and other primitive forms. This language is designed to be as simple as possible in order to focus on the creation and usage of classes and objects. Figure 1 shows our source language syntax, which includes class definition, instantiation with `new`, method and field definition and selection, a `match` construct that generalizes `instanceof` and casts, the lambda calculus, `let`, and `letrec`. We omit mutable fields, inheriting method bodies, and other constructs in order to focus on the basic object-oriented mechanisms of classes, dispatch, and instance checks. The omitted constructs can be added in standard ways [25].

The most central construct of our language is the class definition expression, which defines the set of fields and methods as usual, and binds the newly defined class to a name x that can be used in the expression e. This binding is recursive so that methods can refer to the type of the containing class. We bind a name to the class immediately for this purpose and so that the type system can use the class's name to reason about its identity. However, class creation is dynamic: class definition expressions may occur inside a function or method, and a new class with a distinct run-time tag is created each time the function is called. The created class is also first-class: it has type $\texttt{class}\ x\ \{\overline{f:\tau}, \overline{m:\tau}\}$ and can be passed to a function or method, or bound to another variable y. A singleton type mechanism [17] could be added to support strong reasoning about equality of classes, but for simplicity we do not explore this here.

We illustrate the language with a simple counter class that increments a value when the `inc` method is called:

```
1 class Counter {
2   val:int,
3   inc:unit->Counter obj=λ_:unit.new(Counter;this.val+1)
4 } in let oneC = new (Counter; 1)
5   in let twoC = oneC.inc() ...
```

In the example, `val` is a field that will be initialized when a `Counter` is created. `inc` is a method; the difference between methods and fields is that a method is given a definition in the class body (typically using a function, although our semantics does not enforce this), and the special variable `this` is in scope within the body of the method.

As mentioned earlier, classes and objects in the source language are immutable for the sake of simplicity. Therefore, instead of actually updating the `val` field, the `inc` method

simply returns a new object with its `val` field equal to the receiver object's `val` field plus one. The return type of the `inc` method is `Counter obj`, which is the type of objects that are instances of the `Counter` class.

A subclass can be created using a variant of the class expression that specifies the superclass x. We require the use of a variable rather than an expression in the `extends` clause to allow static reasoning about the identity of the class being extended. A subclass can define additional members, or define members that are subtypes of the corresponding members in the superclass, thus following typical depth and width subtyping rules. The type of a subclass includes the name of the class it extends.[1]

To continue the simple counter example, consider a possible subclass that can reset the counter to zero:

```
1 class RCounter extends Counter{
2   ...
3   reset:unit->RCounter obj=λ_:unit.new(RCounter;0)
4 } in e
```

The $\texttt{new}(x; \overline{e})$ expression takes a variable x representing a class and a list of expressions. Executing the `new` expression instantiates an object of class x with the input expressions as initial values for the fields of the object. Alternative constructors with pre-defined field values can be created by wrapping `new` in a function or method.

An important piece of the source language is the `match` expression. This checks whether or not the object passed in as the first argument is an instance of the class named in the second argument. If the instance check passes, then the variable x_2 is bound to the object originally passed in, but x_2 is typed according to the class that was checked against, and expression e_2 is executed. Expression e_3 is executed in the default case where the match fails.

The match expression could be expressed equivalently in terms of `instanceof`, downcasting, and an `if` statement:

$$\begin{aligned}&\texttt{match}(e_1; x; y.e_2; e_3) = \\ &\quad \texttt{if}(e_1\ \texttt{instanceof}\ x)\quad \texttt{then let}\ y = (x)e_1\ \texttt{in}\ e_2\quad \texttt{else}\ e_3\end{aligned}$$

Similarly both `instanceof` and downcasting can be emulated by `match`:

$$\begin{aligned}e\ \texttt{instanceof}\ x &= \texttt{match}(e; x; \_.\texttt{true}; \texttt{false}) \\ \texttt{cast}(e; x) &= \texttt{match}(e; x; y.y; \texttt{err})\end{aligned}$$

where `err` indicates an execution error (e.g. throwing an exception) upon a failed cast. The `err` construct demonstrates that `match` is safer than downcasts because a default case must be given.

Because class creation expressions can be nested in the body of a function or method, they can be executed more than once – and each time the expression is executed, a different class (i.e. a class with a different static type and run-time tag) is generated. This means that `match` cannot be implemented using a lookup in a static class table. Instead, as the next section will show, we need a dynamic environment tracking all the classes that have been created, along with their relationships to one another, in order to define the semantics of `match`.

[1] Although our system supports subtyping, we omit inheritance (e.g. of code in method bodies) in order to focus our attention on a simple type-theoretic formalism that captures the essence of tags.

2.1 Applications of First-Class Classes

There are a number of useful applications of first-class classes (and thus of dynamically generated hierarchical tags), a few of which we sketch here.

Encapsulation. One of the simplest applications is ensuring that data private to an object stays private. For example, imagine a map made up of a linked list of entries. We would like to ensure that no entry is shared by multiple maps. We can do this by generating a fresh `Entry` class for each map object. This idiom is similar to uses of generative functors in languages such as Standard ML [16].[2] In our language, it looks like this:

```
let MyMap =
  class Entry {
    key:int,
    value:int,
    next:Entry obj
  } in
    class Map extends IMap {
      head:Entry obj,
      put: int->int->Map obj = e_put,
      get: int->int = e_get
    } in Map
in ...
```

Each time the `let` statement is executed, we generate a new `Map` class, and each `Map` class has a corresponding `Entry` class used to hold its private data. The type system will not allow one map to contain entries defined by another map. Because all maps implement the same `IMap` interface, however, clients can store and manipulate maps uniformly.

Memento. A similar approach can be used to implement the Memento design pattern [12], in which an originator object externalizes its state as a separate memento object. The originator object's state can later be restored using the memento. We may want to ensure that mementos created by different objects are not mixed up. To do this, we generate a fresh `MyMemento` class for each originator object, ensuring that it is a subclass of the well-known `Memento` class. To externalize its state, each originator then creates memento objects using its own private `MyMemento` class. When the originator's state is restored, we check that the memento used for restoration is of the private class, ensuring that the memento was created for this object and not some other object. This example differs from the previous one in that mementos are deliberately exposed to clients, and in that checking for the right class is done dynamically using match rather than statically using the object type:

```
class Memento {intState:int} in
// the code below may be in a loop or function
let Originator =
class MyMemento extends Memento
  {intState:int}
 in
  class Originator {
   myState:int
   saveToMemento: T -> Memento obj =
     λx:int.new(MyMemnto; this.myState),
   restoreFromMemento:Memento obj->Originator =
     λm:Memento obj.
       match(m;MyMemnto;
         x.new(Originator;x.intState);
         raise RuntimeException)
  in Originator
in ...
```

[2] An alternative would be to use ownership types [7].

Mixins. Another application area for dynamically created tags is mixin compositions. According to Bracha and Cook, "A *mixin* is an abstract subclass; i.e. a subclass definition that may be applied to different superclasses to create a related family of modified classes. For example, a mixin might be defined that adds a border to a window class; this mixin could be applied to any kind of window to create a bordered-window class" [4].

Achieving the full benefits of mixins – and indeed, properly implementing the missing method bodies below – requires inheritance as well as tagged objects. Prior work on mixins shows how to obtain expressive code reuse via inheritance [2]. Our work is not intended to replace this work, but rather complement prior results by adding tagging support in a setting with first-class classes. In this section, we omit inheritance –which can be added back through standard encodings – in order to focus on the benefits of these added features.

The library code below illustrates a version of the bordered window example. Class Window defines a draw method (and perhaps other methods, not shown). We define a mixin as a function[3] that accepts a subclass of Window and extends that class with border functionality. A similar mixin can make an arbitrary window scrollable.

```
class Window {
    draw:unit->unit = /* draw a blank window */
}

let makeBordered =
    λwc: class WinClass extends Window { ... } .
        class BorderedWinClass extends wc {
            draw:unit->unit = /* draw a bordered window */
        } in BorderedWinClass

let makeScrollable =
    λwc: class WinClass extends Window { ... } .
        class ScrollableWinClass extends wc {
            draw:unit->unit =
                /* draw scrollbars and scroll contents */
        } in ScrollableWinClass
```

Now, in the application code below, we can create a custom application window as a subclass of Window. We can then make it bordered, conditional on whether the user's preferences specify a border. This can be a run-time check, illustrating the dynamic nature of class composition in this example. We can also mix in the scrollable property, which will be used when creating large windows. Later, we can define operations such as screen capture that should behave differently for different kinds of windows, using a match statement.[4]

```
class AppWindow extends Window { ... }

let WinCls =
    if (/* check if user wants a border */)
        makeBordered(AppWindow)
    else
        AppWindow

let BigWinCls = makeScrollable(winCls) in
let smallWin = new WinCls(...) in
let bigWin = new BigWinCls(...) in
let screenCap = λw:Window .
    match(w,BigWinCls,
```

[3] Similar to ML functor [16].

[4] We could also have made `screenCap()` a method of `Window`, but only if we anticipated it in the library. If we only think of it when we write the application, it is typically not feasible to add new methods to the library, and so using `match` statements in place of dispatch is critical.

$n ::= x \mid c \mid \texttt{fst}(n) \mid \texttt{unfold}(n)$

$e ::= \texttt{newtag}[\tau] \mid \texttt{subtag}[\tau](n) \mid \texttt{new}(n; e) \mid \texttt{match}(e; n; x.e; e) \mid \texttt{extract}(e) \mid \lambda x{:}\tau.e \mid e\ e$
$\quad \mid \{\overline{f = e}\} \mid e.f \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{letrec}\ x = e\ \texttt{in}\ e \mid \texttt{fold}[t.\tau](e) \mid \texttt{unfold}(e)$
$\quad \mid \langle e, e\rangle \mid \texttt{fst}(e) \mid \texttt{snd}(e) \mid \langle\rangle \mid n$

$\tau ::= Tag(\tau) \mid \texttt{tagged}\ n \mid \Pi_{x:\tau}.\tau \mid \{\overline{f : \tau}\} \mid \mu t.\tau \mid \Sigma_{x:\tau}\tau \mid \top \mid t$

$Tag(\tau) ::= \tau\ \texttt{tag} \mid \tau\ \texttt{tag extends}\ n$

$\Gamma ::= \epsilon \mid \Gamma,\ x : \tau$

$\Sigma ::= \epsilon \mid \Sigma,\ c \sim \tau$

$\Delta ::= \epsilon \mid \Delta,\ t <: t$

Figure 2 Core Language Syntax.

```
14          bw./*adjust capture for scrolling windows*/),
15          /*default screen capture*/)
16  in screenCap(bigWin); /* the match above succeeds */
17  screenCap(smallWin) /* the match above will default*/
```

Discussion: modules and dynamic class loading. The mixin example above is reminiscent of functors in Standard ML [16]. Functors can also be used to implement a type that builds on another type, which is bound later when the functor is applied. This suggests that our theory can be used to model the combination of classes and advanced module systems – and in fact, our implementation in Wyvern, discussed in Section 5, combines these features. We believe that our theory might also be useful for exploring the semantics of dynamic class loading, for similar reasons, but we have not yet investigated this connection.

3 A Hierarchical Tagging Language

We would like to find a way to explain the semantics of the language defined in the previous section in terms of type theory. Figure 2 defines the syntax of a core language for doing so. We start with constructs required for traditional object encodings based on recursive records: the lambda calculus, record creation and projection, ordinary and recursive let bindings, and iso-recursive types with fold and unfold constructs. Our function types are dependent so that we can define functions that act like functors, accepting a tag as an argument and producing a subtag of the argument as a result. We include units and a top type.

We also include dependent sum types, because our translation for the source-level language will translate a class into a pair containing the class's tag and a function to construct objects of that class. Since objects of the class have the class's tag as part of type, the constructor function must also; thus the pair has the type of a dependent sum.

These constructs are more than sufficient to encode record-based objects. However, we need more to allow us to model the classes and `match`-based instance testing constructs of the OO source language, while also supporting a type safety property.

As described in the introduction, there is a natural parallel between class types, class generation and match in our surface language, and extensible sums, tag generation and tag testing in functional programming languages such as Standard ML. We therefore add mechanisms for generating tags, tagging values, testing against tags, and extracting values from

tagged objects. Our presentation is inspired both by Glew's work on modeling hierarchical tags [13], and by the coverage of symbols and dynamic classification in Harper's book [14].

The most basic way to create a tag is using the expression $\texttt{newtag}[\tau]$ that, when executed, will generate a new tag (let's call it by the name n). The new tag can be used to create a *tagged value* using the expression $\texttt{new}(n; e)$, which takes the value to which e reduces and tags it with the tag n. In this expression, e must have the type τ that was associated with tag n when n was created. The type of the tagged expression is $\texttt{tagged}\ n$; thus the static type system tracks not only the fact that the expression is tagged, but what tag it is tagged with. In this respect, we model OO type systems that track the class of objects, but differ from prior work on modeling tags because they did not include the tag as part of the type [13, 14, 16].

While analogies can be drawn between tag creation and class creation, the type of τ is not restricted to be a record as in the source language. Rather, any arbitrary type τ is allowed to be tagged.

Following Glew [13], we provide a construct for extending an already created tag with a subtag. The construct $\texttt{subtag}[\tau](n)$ creates a new tag as a subtag of n. Note that tags to be extended in this rule are represented by names n; most prominently, names include variables x in the source code. Using names supports static reasoning about tag identity, just as in the source language we required the subclass to be specified as a variable. In addition to variables, names can be the first element of a dependent pair that is also represented by a name (we use this in our encoding – in principle we could include the second element as well), or an unfolding of a name, or they can be tag values c generated at run time. Here c is like a location in formalisms of references, and similar to locations, it cannot appear in the source code. We use Σ to associate tag values with types ($c \sim \tau$); this is analogous to a store type. Names *cannot* be arbitrary expressions; this restriction means we do not have to evaluate expressions in the type system, avoiding the intractability that this entails in the presence of recursion.

Tag testing is done through the $\texttt{match}(e_1; n; y.e_2; e_3)$ expression, which has the same structure as the `iftagof` construct from prior work in type theory [16, 13]. This expression takes in a tagged expression e_1, reduces it to a tagged value, and compares it to the tag n. Upon a successful match, the tagged value is bound to y, and the expression e_2 can use y as if it were of type $\texttt{tagged}\ n$. Like its source level equivalent, `match` requires there to be a failure branch, e_3, that is evaluated upon an unsuccessful tag test.

To extract an expression from a tag, $\texttt{extract}(e)$ is used. If e is a value tagged with n, then $\texttt{extract}(e)$ unwraps the tagged value and exposes the internal contents e.

Hierarchical tagging can be used to emulate a makeshift sum type. For example, the sum type $\texttt{int} + \top$, which would correspond to nullable `int`s, can be emulated in a local scope through tags:

$$
\begin{array}{l}
\texttt{let intOption} : \top\ \texttt{tag} = \texttt{newtag}[\top]\ \texttt{in} \\
\texttt{let none} : \top\ \texttt{tag extends intOption} = \texttt{subtag}[\top](\texttt{intOption})\ \texttt{in} \\
\texttt{let some} : \texttt{int tag extends intOption} = \texttt{subtag}[\texttt{int}](\texttt{intOption})\ \texttt{in} \\
\texttt{let}\ z : \texttt{tagged}\ \mathit{none} = \texttt{new}(\texttt{none}; \langle\rangle)\ \texttt{in} \\
(\lambda x\colon \texttt{tagged intOption}. \\
\quad \texttt{match}(x; \texttt{some}; y.\texttt{extract}(y) + 1; -1))(z)
\end{array}
$$

This expression would test `none` against `some`, and failing to match, will evaluate to -1. This example demonstrates how these constructs allow one to create extensible sum types in a local scope.

$$\frac{}{\Delta|\Gamma \vdash_\Sigma \tau <: \tau}(\text{ST-Reflexive}) \qquad \frac{\Delta|\Gamma \vdash_\Sigma \tau_1 <: \tau_2 \qquad \Delta|\Gamma \vdash_\Sigma \tau_2 <: \tau_3}{\Delta|\Gamma \vdash_\Sigma \tau_1 <: \tau_3}(\text{ST-Trans})$$

$$\frac{t <: t' \in \Delta}{\Delta|\Gamma \vdash_\Sigma t <: t'}(\text{ST-Amber-1}) \qquad \frac{\Delta, t<:t' \mid \Gamma \vdash_\Sigma \tau <: \tau'}{\Delta|\Gamma \vdash_\Sigma \mu t.\tau <: \mu t'.\tau'}(\text{ST-Amber-2})$$

$$\frac{}{\Delta|\Gamma \vdash_\Sigma \{f_i : \tau_i{}^{i\in 1..n+k}\} <: \{f_i : \tau_i{}^{i\in 1..n}\}}(\text{ST-Record-1})$$

$$\frac{\text{for each } i \quad \Delta|\Gamma \vdash_\Sigma \tau_i <: \tau_i'}{\Delta|\Gamma \vdash_\Sigma \{f_i : \tau_i{}^{i\in 1..n}\} <: \{f_i : \tau_i'{}^{i\in 1..n}\}}(\text{ST-Record-2})$$

$$\frac{\{l_j : \rho_j{}^{j\in 1..n}\} \text{ is a permutation of } \{f_i : \tau_i{}^{i\in 1..n}\}}{\Delta|\Gamma \vdash_\Sigma \{l_j : \rho_j{}^{j\in 1..n}\} <: \{f_i : \tau_i{}^{i\in 1..n}\}}(\text{ST-Record-3})$$

$$\frac{\Delta|\Gamma \vdash_\Sigma \tau_3 <: \tau_1 \qquad \Delta|\Gamma, x : \tau_3 \vdash_\Sigma \tau_2 <: \tau_4}{\Delta|\Gamma \vdash_\Sigma \Pi_{x:\tau_1}.\tau_2 <: \Pi_{x:\tau_3}.\tau_4}(\text{ST-App})$$

$$\frac{\Gamma \vdash_\Sigma n : \tau \texttt{ tag extends } n'}{\Delta|\Gamma \vdash_\Sigma \texttt{tagged } n <: \texttt{tagged } n'}(\text{ST-Tag-1})$$

$$\frac{\Delta|\Gamma \vdash_\Sigma \texttt{tagged } n <: \texttt{tagged } n'}{\Delta|\Gamma \vdash_\Sigma \tau \texttt{ tag extends } n <: \tau \texttt{ tag extends } n'}(\text{ST-Tag-2})$$

$$\frac{}{\Delta|\Gamma \vdash_\Sigma \tau \texttt{ tag extends } n <: \tau \texttt{ tag}}(\text{ST-Tag-3})$$

Figure 3 Core Language Subtyping Rules.

3.1 Subtyping

In our source language, subclasses define subtypes, and so our core language needs to define subtyping as well. The subtyping judgment is of the form $\Delta|\Gamma \vdash_\Sigma \tau <: \tau$. The extra context Δ for the subtyping judgment stores subtyping relations between type variables and is used in the Amber Rule (ST-Amber-1 and ST-Amber-2) for subtyping of recursive types [6].

Most of the subtyping rules in Figure 3 are standard: subtyping is reflexive and transitive. Recursive types follow the Amber Rule, while records support width and depth subtyping and allow permutation. Finally, we use the standard contravariant rule for dependent function types.

Subtyping between tagged types is nominal, and follows the declared subtagging relationship as expected: that is, `tagged` n is a subtype of `tagged` n' if the type of n reveals that it is a direct subtag of n'. We get transitivity of subtyping on tagged types from the general subtype transitivity rule.

The most interesting rules define the subtyping relation between the types of first-class tags themselves (as opposed to tagged types). If we have a function that accepts a first-class tag parameter of type τ `tag extends` n', what tags do we want it to accept? Intuitively, it should be acceptable to pass an actual argument of type τ `tag extends` n, where n is a subtag of n'; this loses a bit of information, but does not create any unsafe situations. Furthermore, if the function accepts a parameter of type τ `tag` (i.e. a tag with no known supertag) it should be fine to pass it an actual argument that *does* have a supertag, because none of the operations in our core language depend on a tag having no supertag.

On the other hand, it would be unsound to allow the type being wrapped by the tag to vary. This is because tags are like references: values flow into a tagged value when it is created, and they flow out when the `extract` operation is used, so the types of first-class tags cannot be either covariant or contravariant in the type being wrapped.

We can now see how to express the essence of the mixin example shown earlier can be expressed using the core language:

$\texttt{let}\ \mathit{base} : \texttt{int tag} = \texttt{newtag[int]}\ \texttt{in}$

$\texttt{let}\ \mathit{mixin} : \Pi_{c:\texttt{int tag}}.\texttt{int tag extends}\ c = \lambda c{:}\ \texttt{int tag}\ .\ \texttt{subtag[int]}(c)\ \texttt{in}$

$\texttt{let}\ \mathit{sub} : \Pi_{c:\texttt{int tag}}.\texttt{int tag extends}\ c = \lambda c{:}\ \texttt{int tag}\ .\ \texttt{subtag[int]}(c)\ \texttt{in}$

$\texttt{let}\ c : \texttt{int tag extends}\ \mathit{base} = \texttt{if}\ (\mathit{cond})\ \mathit{sub}(\mathit{mixin}(\mathit{base}))\ \texttt{else}\ \mathit{sub}(\mathit{base})\ \texttt{in}$

$\texttt{let}\ x : \texttt{tagged}\ c = \texttt{new}(c; 5)\ \texttt{in}$

...

Here we have a tag *base* and two functions that create subtags of it, one of which is intended to be used as a mixin. Based on some dynamic condition *cond*, the program will create the tag c to either be a direct subtag of base or a transitive one. Because of the subtyping rules between tags, we can pass either *base* or $\mathit{mixin}(\mathit{base})$ to *sub* safely. Of course, the mixin above adds neither functionality nor state to the object, but the point of the example is getting the tags and the typing right; adding a standard encoding of inheritance will allow the mixin functionality to work.

3.2 Typing

The typing judgment for the core language is of the form:

$$\Gamma \vdash_\Sigma e : \tau$$

This judgment uses two different contexts. Γ is the standard type context, while Σ is introduced for technical reasons: it is like a store type, but rather than track the types of references, it tracks the types of generated tag values at run time, facilitating the proof of type safety (see the supplementary material [20]) in the same way that store types do in formalizations of imperative languages.

The typing rules for our core language are mostly standard; Figure 4 shows the rules for the constructs we introduced, as well as a couple of others that are more involved. The remaining rules are the standard ones for the lambda calculus, and so are omitted here. Our function application rule is the standard one, except that since we have a dependent function type, we must substitute the actual argument for the formal parameter in the result type. Since names, but not arbitrary expressions, can appear in types, applying the substitution to the result type is only defined when the argument e_2 is a name, or else when the variable x being substituted for is not free in the type τ' (note that if substitution is not defined then the App rule may not be applied).

Our rules include a subsumption rule that can be applied at any point, and a rule that looks up the type of a tag value c in the tag store type Σ (see the dynamic semantics, below, for how tag values are generated). Defining a new tag for τ values yields the obvious type τ `tag`. The rule for creating subtags is similar, but the premises must check that the name n given as a supertag is actually a tag, and that the type τ' being tagged is a subtype of the type τ that is tagged by the supertag. The `new` expression is well-typed as tagged by n if n is a tag and the expression used is a subtype of the type associated with the n tag.

$$\frac{\Gamma \vdash_\Sigma e_1 : \Pi_{x:\tau}.\tau' \qquad \Gamma \vdash_\Sigma e_2 : \tau}{\Gamma \vdash_\Sigma e_1 e_2 : [e_2/x]\tau'}(\textsc{App}) \qquad \frac{\Gamma \vdash_\Sigma e : \tau \qquad \epsilon|\Gamma \vdash_\Sigma \tau <: \tau'}{\Gamma \vdash_\Sigma e : \tau'}(\textsc{Sub})$$

$$\frac{\Sigma(c) = \tau}{\Gamma \vdash_\Sigma c : \tau}(\textsc{CVar}) \qquad \frac{}{\Gamma \vdash_\Sigma \texttt{newtag}[\tau] : \tau\ \texttt{tag}}(\textsc{Cls-I})$$

$$\frac{\Gamma \vdash_\Sigma n : Tag(\tau) \qquad \epsilon|\Gamma \vdash_\Sigma \tau' <: \tau}{\Gamma \vdash_\Sigma \texttt{subtag}[\tau'](n) : \tau'\ \texttt{tag extends}\ n}(\textsc{CCls-I})$$

$$\frac{\Gamma \vdash_\Sigma n : Tag(\tau) \qquad \Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \texttt{new}(n; e) : \texttt{tagged}\ n}(\textsc{Tag-I})$$

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma e_1 : \texttt{tagged}\ n' \quad \Gamma \vdash_\Sigma \texttt{tagged}\ n' \updownarrow \texttt{tagged}\ n \\ \Gamma, x : \texttt{tagged}\ n \vdash_\Sigma e_2 : \tau \qquad \Gamma \vdash_\Sigma e_3 : \tau\end{array}}{\Gamma \vdash_\Sigma \texttt{match}(e_1; n; x.e_2; e_3) : \tau}(\textsc{Match})$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{tagged}\ n \qquad \Gamma \vdash_\Sigma n : Tag(\tau)}{\Gamma \vdash_\Sigma \texttt{extract}(e) : \tau}(\textsc{Extract})$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \tau_1 \qquad \Gamma \vdash_\Sigma e_2 : [e_1/x]\tau_2}{\Gamma \vdash_\Sigma \langle e_1, e_2 \rangle : \Sigma_{x:\tau_1}\tau_2}(\Sigma\text{-I}) \qquad \frac{\Gamma \vdash_\Sigma e : \Sigma_{x:\tau_1}\tau_2}{\Gamma \vdash_\Sigma \texttt{fst}(e) : \tau_1}(\Sigma\text{-E}_1)$$

$$\frac{\Gamma \vdash_\Sigma e : \Sigma_{x:\tau_1}\tau_2}{\Gamma \vdash_\Sigma \texttt{snd}(e) : [\texttt{fst}(e)/x]\tau_2}(\Sigma\text{-E}_2)$$

Definition: $\Gamma \vdash_\Sigma \tau_1 \updownarrow \tau_2 = \exists \tau_p \ .\ \epsilon|\Gamma \vdash_\Sigma \tau_1 <: \tau_p$ and $\epsilon|\Gamma \vdash_\Sigma \tau_2 <: \tau_p$.

Figure 4 Core Language Static Semantics.

One of the more interesting checks comes in the Match typing rule. Here, the tagged expression e_1 should be comparable to the tag n. If e_1 is tagged with n', then n' should be in the same branch of the tag hierarchy as n. This prevents pointless attempts to cast a tagged value to something of an entirely different tag "tree". Formally, this means that `tagged` n and `tagged` n' should have a common supertag in the hierarchy: there exists some n'' such that `tagged` n <: `tagged` n'' and `tagged` n' <: `tagged` n''. We define the $\updownarrow$ notation at the bottom of Figure 4 that specifies the need to go "up and down" the tag hierarchy.

Extracting a value from a tagged expression only requires that e is of type `tagged` n in `extract`(e). If e is tagged with n, then there is no harm in unwrapping e and typing it with the type that n tags.

As mentioned before, dependent sums are in the core language. The translation from the source to core language–shown in the next section–heavily uses dependent sums to convert classes, essentially because the type system must associate the tag generated for a class with the result type of the class constructor, which is also generated for that same class. We use the type abbreviation $\tau_1 \times \tau_2$ as a special case of the dependent sum $\Sigma_{x:\tau_1}.\tau_2$, where τ_2 does not use x. Furthermore, we can use the type abbreviation $\tau_1 \to \tau_2$ as a special case of the dependent function $\Pi_{x:\tau_1}.\tau_2$, where τ_2 does not depend on x.

As in typical dependent type systems, the type `tagged` B is only well formed if B is in the context Γ. This leads to the avoidance problem [15] common in module systems, and requires the type system to use the subsumption rule when a variable goes of scope. For

$$S ::= \epsilon \mid S, c \rightsquigarrow p \qquad p ::= \varepsilon \mid c \rightsquigarrow p \qquad \frac{}{c \in c \rightsquigarrow p} \qquad \frac{c \in p \qquad c \neq c'}{c \in c' \rightsquigarrow p}$$

Notation: $S, x = S, x \rightsquigarrow \varepsilon$

Figure 5 Hierarchical Store Definition and Rules.

example, consider the following:

$\texttt{let } A : \texttt{int tag} = \texttt{newtag[int] in}$
$\texttt{let } x : \texttt{tagged } A =$
$\quad \texttt{let } B : \texttt{int tag extends } A = \texttt{subtag[int]}(A) \texttt{ in}$
$\quad \texttt{let } f : \Pi_{C:\texttt{int tag}}.\texttt{tagged } C = \lambda C{:}\texttt{int tag} \,.\, \texttt{new}(C; 1) \texttt{ in}$
$\quad \texttt{let } y : \texttt{tagged } B = f(B)$
$\quad \texttt{in } y$
$\ldots$

Here f is a dependent function type, so when we invoke $f(B)$ we know that the result (bound to y) has type `tagged` B. On the other hand, if we then bind y to x in a scope where B is not visible (e.g. the second line of code above), we have to treat x as having the supertype `tagged` A because we must avoid mentioning B in the type.

3.3 Dynamics

The dynamics of the core language utilizes a hierarchical store (Figure 5) that represents all tags that have been generated in the execution of the program so far. Using a store to dynamically track tags is necessary because if a tag is created by a function, the function may be called an arbitrary number of times and thus generate an arbitrary number of tags. In the hierarchical store, a hierarchy is represented by a set of paths, each starting from a tag c and pointing to a (possibly empty) sub-path containing that tag's supertags, written as $c \rightsquigarrow p$. The empty path is ε and the end of a path $c \rightsquigarrow \varepsilon$ is shortened to c for notational clarity.

Operational semantics in the core language are expressed by small-step rules and value judgments. Value judgments are of the form

$$S \mid e \texttt{ val}$$

which states that e is a value in the operational semantics under the context of the hierarchical runtime store S. The transition relations are of the form

$$S \mid e \mapsto S' \mid e'$$

which states that e transitions to e' while potentially extending the store S to some S'. In the case the store is not modified, $S = S'$.

The expressions `newtag` and `subtag` create global tags into the hierarchical store. Top-level tags, which are of the form c, are created through evaluation of `newtag`. Note that `newtag` itself is not a value, but instead will evaluate to the fresh tag c.

This expresses the dynamic nature of tag creation in the core language. While tags were represented by local variables in the statics and syntax, the operational semantics

$$\frac{S \mid e \ \texttt{val}}{S \mid \texttt{new}(n; e) \ \texttt{val}}(\textsc{New-V}) \qquad \frac{}{S \mid \lambda x\colon \tau.e \ \texttt{val}}(\textsc{Lam-V})$$

$$\frac{c \in S}{S \mid c \ \texttt{val}}(\text{C-V}) \qquad \frac{\text{for each } i \ \ S \mid e_i \ \texttt{val}}{S \mid \{\overline{f_i = e_i}\} \ \texttt{val}}(\textsc{Rec-V})$$

$$\frac{S \mid e_1 \ \texttt{val} \qquad S \mid e_2 \ \texttt{val}}{S \mid \langle e_1, e_2 \rangle \ \texttt{val}}(\textsc{Prod-V}) \qquad \frac{}{S \mid \langle\rangle \ \texttt{val}}(\textsc{Unit-V})$$

Figure 6 Core Language Value Judgments.

generate fresh global tags at runtime and store them in a hierarchy separate from the type hierarchy. Nonetheless, there is a direct relationship between the hierarchy created during static checking and the hierarchy created at runtime. Any tag n that extends an n' in the static tag hierarchy should have a corresponding c that contains c' (corresponding to n') in its path in the runtime hierarchical store.

Therefore `subtag` also creates a global tag c' and stores it into the hierarchical store, keeping track of the transitive tag path $c' \rightsquigarrow (c \rightsquigarrow p)$ that goes from c' to its parent tag c and on to the root of the hierarchy via the (possibly empty) path p. This leads to the store having the shape of an inverted tree, where each node points to its parent in the store. To check if a tag c is a subtag of another tag c', the dynamics will just follow the path from c back to the root, searching for c' along the way. The dynamic semantics express this search in the rules for $c \in p$.

The $\texttt{match}(e_1; c'; y.e_2; e_3)$ expression traverses the hierarchy to check if a tag is a valid subtag of the provided tag. Given a tagged value $e_1 = \texttt{new}(c; e)$, the dynamics of `match` check if c' is a supertag of c by traversing the path $c \rightsquigarrow p$ and searching for c'. This process is expressed by the relationship $c' \in c \rightsquigarrow p$ defined in Figure 5. A search fails when the path being searched is empty i.e. when $p = \varepsilon$. Upon a successful search, e is substituted for y in e_2. A failed search leads to e_3 instead.

Evaluating `extract` is straightforward. The dynamics do not check whether or not the tag given to `extract` is the same tag in side the `new` expression. Rather, evaluation simply strips off the tag from $\texttt{new}(c; e)$, leaving the untagged expression e. The typing rule for `extract`, together with the constraint that subtags must wrap a subtype of the type their supertags wrap, ensures that the extracted value is a subtype of the expected type.

The value judgments are formalized in Figure 6. Note that the dynamics of the core language use a call-by-value evaluation strategy. Consider the following partial expression:

$$\texttt{let } x = \texttt{newtag}[\texttt{int}] \texttt{ in } e$$

As the `newtag` is evaluated eagerly, only one tag is generated in the global store, since the expression first steps to

$$\texttt{let } x = c \texttt{ in } e$$

where c is a fresh tag in the global hierarchical store. If instead a call-by-name strategy was adopted, then the expression would step to

$$[\texttt{newtag}[\texttt{int}]/x]e$$

$$\frac{c \notin S}{S \mid \texttt{newtag}[\tau] \mapsto S, c \mid c}(\mapsto \textsc{Cls})$$

$$\frac{c' \notin S}{S, c \rightsquigarrow p \mid \texttt{subtag}[\tau](c) \mapsto S, c' \rightsquigarrow (c \rightsquigarrow p), c \rightsquigarrow p \mid c'}(\mapsto \textsc{CCls})$$

$$\frac{S \mid e \mapsto S' \mid e'}{S \mid \texttt{new}(n; e) \mapsto S' \mid \texttt{new}(n; e')}(\mapsto \textsc{New})$$

$$\frac{S \mid e \mapsto S' \mid e'}{S \mid \texttt{match}(e; c; y.e_2; e_3) \mapsto S' \mid \texttt{match}(e'; c; y.e_2; e_3)}(\mapsto \textsc{Match})$$

$$\frac{S, c \rightsquigarrow p \mid e\ \texttt{val} \qquad c' \in c \rightsquigarrow p}{S, c \rightsquigarrow p \mid \texttt{match}(\texttt{new}(c; e); c'; y.e_2; e_3) \mapsto S, c \rightsquigarrow p \mid [\texttt{new}(c; e)/y]e_2}(\mapsto \textsc{MatchSuc})$$

$$\frac{S, c \rightsquigarrow p \mid e\ \texttt{val} \qquad c' \notin c \rightsquigarrow p}{S, c \rightsquigarrow p \mid \texttt{match}(\texttt{new}(c; e); c'; y.e_2; e_3) \mapsto S, c \rightsquigarrow p \mid e_3}(\mapsto \textsc{MatchFail})$$

$$\frac{S \mid e \mapsto S' \mid e'}{S \mid \texttt{extract}(e) \mapsto S' \mid \texttt{extract}(e')}(\mapsto \textsc{Untag1})$$

$$\frac{S \mid e\ \texttt{val}}{S \mid \texttt{extract}(\texttt{new}(\_; e)) \mapsto S \mid e}(\mapsto \textsc{Untag2})$$

Figure 7 Core Language Dynamic Semantics.

which would create a tag for every binding site of x in e – and the tags used to tag values would be different from the tags used as matching targets, causing all matches to fail! Thus the choice of call-by-value is essential.

Evaluation semantics for expressions related to hierarchical tagging can be found in Figure 7. Congruence rules for records, tuples, and let-bindings have been omitted for the sake of brevity.

3.4 Type Soundness

Stated here are the basic type safety theorems for the core language.

Preservation. If $\Gamma \vdash_\Sigma e : \tau$, and $S \mid e \mapsto S' \mid e'$, then $\exists \Sigma' \supseteq \Sigma$ such that $\Gamma \vdash_{\Sigma'} e' : \tau$.
Progress. If $\epsilon \vdash_\Sigma e : \tau$ and $\Sigma \vdash S$, then either $S \mid e\ \texttt{val}$ or $S \mid e \mapsto S' \mid e'$.

The proofs are sketched in the supplementary material [20].

4 Source Language Translation

We would like to define the semantics of our source language. One option is to do so directly, via typing rules and an operational semantics. However, the source language includes a lot of inessential complexity, mainly having to do with the built-in features of classes and methods, all of which are well understood. A source-level semantics would therefore include a fair amount of complexity, but it would be complexity that is not very interesting. Furthermore,

we cannot build directly on previous formal systems, because we are not aware of prior work that includes first-class classes and tag tests in a statically typed setting.

For these reasons–and also considering space constraints–we follow Harper and Stone [16] in defining the semantics of the source language via a translation in to the core language defined in section 3. Our translation has two judgments. Type translation is of the form:

$$\Gamma \vdash \tau \Rightarrow \tau^\dagger$$

where τ is a type in the source language and $\tau^\dagger$ is a type in the core language. Expression translation is of the form

$$\Gamma \vdash e \Rightarrow e^\dagger : \tau^\dagger$$

where e is an expression in the source language and $e^\dagger$ along with $\tau^\dagger$ are respectively an expression and its type in the core language. The translation is primarily concerned with converting classes and objects to the core language that does not have such expressions.

Our translation is based on Bruce et al.'s encoding [5]. In this approach, an object is encoded as a record through the use of recursive types. This encoding is commonly used when translating expressions in an object calculus into a typed lambda calculus. However, the encoding in Bruce et al. leaves out the ability to perform instance checks on an object at run time, as a naive recursive record translation discards the relation between the object and the class it instantiates. If there is no tag involved, then it is impossible to create an object and use `match` to check if it is an instance of a class.

Our solution to the problem is to wrap the encoded object in a `tagged` n, where n is a tag representing a class in the source language. Then match statements in the source language can be translated directly into matches against the corresponding tag in the core language. On the other hand, to invoke a method or access a field, our translation will need to first use `extract` to get a record from the tagged value, then select the appropriate method or field and invoke or read it.

The translation of classes is more complicated. A class must encapsulate both the tag that tags its instances, and a constructor that is used to create instances. We therefore translate classes into a pair consisting of a tag and a constructor function that takes initial values for the fields and generates a tagged value. Because the type of the constructor depends on the value of the tag, we do not use an ordinary pair but rather a dependent sum type. Finally, methods and fields within the class can refer to the class itself, so we wrap the pair in a recursive type.

We can now understand the type translation rules in Figure 8. An object type is translated by looking up the type of the class, recursively translating the class type into a recursive dependent pair type representing the class, and then generating a core type `tagged` `fst`(`unfold`(x)), a value tagged with the first element of the unfolded recursive pair. The rule for translating function types is simple: it just recursively translates the types of the function argument and result.

The translation for class types begins in the premises by recursively translating the field and method types. When doing so, the type x being defined must be in scope, because the methods and fields might use that type in their definitions (e.g. when defining a binary method or a recursive data structure). We generate a dependent pair, naming the first element of the pair x_{tag} so we can use it in the second element. The first element is a tag that extends the tag of x' – which will be `fst`(x') in the translation – if a parent class x' was specified. Note that we use [] for optional syntax in the rule – all brackets must be present or absent when the rule is instantiated. We abbreviate the class body as $I(x)$ as the class

Let $I(x)$ be the type interface a given class defines, i.e. $I(x) = \{\overline{f : \tau_f}, \overline{m : \tau_m}\}$ for $\texttt{class}\ x\ I(x)\ \texttt{in}\ e$, and similarly for subclasses. Then let $I(\tau') = \{\overline{f : [\tau'/x]\tau_f}, \overline{m : [\tau'/x]\tau_m}\}$ to represent changing all instances of x in the class interface. We abbreviate $I^\dagger(x) = \{\overline{f : \tau_f^\dagger}, \overline{m : \tau_m^\dagger}\}$.

Type Translation $\Gamma \vdash \tau \Rightarrow \tau^\dagger$

$$\frac{\Gamma(x) = \tau \qquad \Gamma \vdash \tau \Rightarrow \mu t.\Sigma_{x_{\texttt{tag}}:\ldots}(\overline{\tau_f^\dagger} \to \texttt{tagged}\ x_{\texttt{tag}})}{\Gamma \vdash x\ \texttt{obj} \Rightarrow \texttt{tagged}\ \texttt{fst}(\texttt{unfold}(x))}$$

$$\frac{\Gamma \vdash \tau_1 \Rightarrow \tau_1^\dagger \qquad \Gamma \vdash \tau_2 \Rightarrow \tau_2^\dagger}{\Gamma \vdash \tau_1 \to \tau_2 \Rightarrow \tau_1^\dagger \to \tau_2^\dagger}$$

$$\frac{\begin{array}{rl}\forall f_i\colon \tau_{f_i} & \Gamma, x\colon \texttt{class}\ x\ I(x)[\texttt{extends}\ x'] \vdash \tau_{f_i} \Rightarrow \tau_{f_i}^\dagger \\ \forall m_i\colon \tau_{m_i} & \Gamma, x\colon \texttt{class}\ x\ I(x)[\texttt{extends}\ x'] \vdash \tau_{m_i} \Rightarrow \tau_{m_i}^\dagger\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{class}\ x\ I(x)[\texttt{extends}\ x'] \Rightarrow \\ \mu t.\Sigma_{x_{\texttt{tag}}:(I_\tau^\dagger(t)\ \texttt{tag}\ [\texttt{extends}\ \texttt{fst}(\texttt{unfold}(x'))])}(\overline{\tau_f^\dagger} \to \texttt{tagged}\ x_{\texttt{tag}})\end{array}}$$

Figure 8 Source to Core Type Translation.

name x is recursively bound in the body; the notation $I^\dagger(x)$ defined at the top of the figure simply applies the type translation recursively to the elements of the class body. The class body then becomes a record type.

The second element of the dependent pair represents the constructor, which takes as arguments the types of the class's fields, and returns the tagged type. We wrap the entire pair in a recursive type so that the class type is properly bound in the class body.

We turn now to the expression translation rules in Figures 9 and 10. For the sake of brevity, we omit the translation rules for lambdas, function application, and let-bindings; these rules simply translate the component types and expressions recursively. The first rule translates the `new` expression, first by translating the subexpressions, looking up the translated type of the class being instantiated, and invoking the second member of the unfolded pair representing the class (i.e. the constructor function) with the translated arguments. The result is tagged with the class's tag.

The second rule is for method calls. It translates the receiver. Because the receiver must have been an object type in the source language, we know its type will be of the form `tagged fst(unfold(`x`))`; looking up the type of x in the context, we find the underlying record type $I_\tau^\dagger(\tau^\dagger)$, and select the type of the method m. The rule for field reads on the top right is analogous. The rule for match, second from the top on the right, translates all the component parts and creates a core-level `match` statement that extracts the tag from the first component of the unfolded class value.

Finally, the two rules at the bottom translate class expressions. The result of translation is a recursive value, which we implement with a `letrec` and a `fold`. We construct the pair with a new tag holding the translated record type, and a constructor function that uses another `letrec` to bind `this` to a tagged value that wraps the record itself. The variant on the right is identical to that on the left, except that a subtag is created.

Let $I(x)$ be the type interface a given class defines, i.e. $I(x) = \{\overline{f : \tau_f}, \overline{m : \tau_m}\}$ for $\texttt{class}\ x\ I(x)\ \texttt{in}\ e$, and similarly for subclasses. Then let $I(\tau') = \{\overline{f : [\tau'/x]\tau_f}, \overline{m : [\tau'/x]\tau_m}\}$ to represent changing all instances of x in the class interface. We abbreviate $I^\dagger(x) = \{\overline{f : \tau_f^\dagger}, \overline{m : \tau_m^\dagger}\}$. Finally, $I(\tau)[f]$ yields the type associated with field f in the interface $I(\tau)$.

Expression Translation $\Gamma \vdash e \Rightarrow e^\dagger : \tau^\dagger$

$$\frac{\forall e_i \in \overline{e} \quad \Gamma \vdash e_i \Rightarrow e_i^\dagger : \tau_i^\dagger \qquad \Gamma(x) = \tau \qquad \Gamma \vdash \tau \Rightarrow \mu t.\Sigma \ldots}{\Gamma \vdash \texttt{new}(x; \overline{e}) \Rightarrow (\texttt{snd}(\texttt{unfold}(x)))(\overline{e^\dagger}) : \texttt{tagged}\ \texttt{fst}(\texttt{unfold}(x))}$$

$$\frac{\begin{array}{l}\Gamma \vdash e \Rightarrow e^\dagger : \texttt{tagged}\ \texttt{fst}(\texttt{unfold}(x)) \qquad \Gamma(x) = \tau \\ \Gamma \vdash \tau \Rightarrow \tau^\dagger \qquad \tau^\dagger = \mu t.\Sigma_{x_{\texttt{tag}} : (I_\tau^\dagger(t)\ \texttt{tag}\ \ldots)} \cdots\end{array}}{\Gamma \vdash e.m \Rightarrow \texttt{extract}(e^\dagger).m : I_\tau^\dagger(\tau^\dagger)[m]}$$

$$\frac{\begin{array}{l}\forall m_i : \tau_{m_i} = e_{m_i} \qquad \Gamma, x : \texttt{class}\ x\ I(x) \vdash e_{m_i} \Rightarrow e_{m_i}^\dagger : \tau_{m_i}^\dagger \\ \Gamma \vdash \texttt{class}\ x\ I(x) \Rightarrow \tau_x^\dagger \qquad \tau_x^\dagger = \mu t.\Sigma_{x_{\texttt{tag}} : (I_\tau^\dagger(t)\ \texttt{tag})} \cdots \\ \Gamma, x : \texttt{class}\ x\ I(x) \vdash e \Rightarrow e^\dagger : \tau^\dagger\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{class}\ x\ \{\overline{f : \tau_f}, \overline{m : \tau_m = e_m}\}\ \texttt{in}\ e \Rightarrow \\ \texttt{letrec}\ x = \texttt{fold}[\tau_x^\dagger](\ \langle \texttt{newtag}[I_\tau^\dagger(t)], \quad \lambda \overline{y} : \overline{\tau_f^\dagger}. \\ \qquad \texttt{letrec this} = \quad \texttt{let}\ o = \{\overline{f = y}, \overline{m = e_m^\dagger}\}\ \quad \texttt{in new}(\texttt{fst}(\texttt{unfold}(x)); o) \\ \qquad \texttt{in this}\rangle)\ \quad \texttt{in}\ e^\dagger \quad : \tau^\dagger\end{array}}$$

Figure 9 Source Language to Core Language Expression Translation (Part 1 of 2).

Properties. Note that the translation given above completely defines the semantics for the source language, following the style of semantics given by Harper and Stone for Standard ML [16]. For example, typechecking a source program succeeds exactly if the translation rules apply and the result of translation is well-typed in the target language. This means that there is no separate type safety theorem to prove for the source language; the source language is type safe because it is defined by translation into a type safe target language.

Discussion. Our translation motivates an interesting feature of our core language: the separation of `extract` from `match`. A more obvious choice would have been to combine these, providing a $\texttt{match}(e; n; x.e; e)$ statement that binds x to the extracted value when the match succeeds. This choice works well for *non*-hierarchical tags, and indeed is the construct used in conventional sum types as well as ML's `exn` type and Harper's classified types [14]. It does not work for *hierarchical* tags, however. To see why, imagine we have a three-level hierarchy, with C a subtag of B and B a subtag of A. We can create a `tagged` C and treat it, by subsumption, as a `tagged` A. If we match against B, we do not want to immediately extract the contents, because we want to not forget that it is really a C in case we try to

$$\frac{\Gamma \vdash e \Rightarrow e^\dagger : \texttt{tagged fst}(\texttt{unfold}(x)) \qquad \Gamma(x) = \tau \qquad \Gamma \vdash \tau \Rightarrow \tau^\dagger \qquad \tau^\dagger = \mu t.\Sigma_{x_{\texttt{tag}}:(I^\dagger_\tau(t)\ \texttt{tag}\ \ldots)} \cdots}{\Gamma \vdash e.f \Rightarrow \texttt{extract}(e^\dagger).f : I^\dagger_\tau(\tau^\dagger)[f]}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow e_1^\dagger : \tau_1^\dagger \qquad \Gamma \vdash e_3 \Rightarrow e_3^\dagger : \tau^\dagger \qquad \Gamma, y : x\ \texttt{obj} \vdash e_2 \Rightarrow e_2^\dagger : \tau^\dagger}{\begin{array}{l}\Gamma \vdash \texttt{match}(e_1; x; y.e_2; e_3) \Rightarrow \\ \texttt{match}(e_1^\dagger; \texttt{fst}(\texttt{unfold}(x)); z.[z/y]e_2^\dagger; e_3^\dagger)\ : \tau^\dagger\end{array}}$$

$$\frac{\begin{array}{l}\forall m_i : \tau_{m_i} = e_{m_i} \qquad \Gamma, x : \texttt{class}\ x\ I(x)\ \texttt{extends}\ x' \vdash e_{m_i} \Rightarrow e^\dagger_{m_i} : \tau^\dagger_{m_i} \\ \Gamma \vdash \texttt{class}\ x\ I(x)\ \texttt{extends}\ x' \Rightarrow \tau^\dagger_x \\ \tau^\dagger_x = \mu t.\Sigma_{x_{\texttt{tag}}:(I^\dagger_\tau(t)\ \texttt{tag}\ \ldots)} \cdots \\ \Gamma, x : \texttt{class}\ x\ I(x)\ \texttt{extends}\ x' \vdash e \Rightarrow e^\dagger : \tau^\dagger\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{class}\ x\ \{\overline{f : \tau_f}, \overline{m : \tau_m = e_m}\}\ \texttt{extends}\ x'\ \texttt{in}\ e \Rightarrow \\ \texttt{letrec}\ x = \texttt{fold}[\tau^\dagger_x](\quad \langle \texttt{subtag}[I^\dagger_\tau(t)](\texttt{fst}(\texttt{unfold}(x'))),\quad \lambda \overline{y} : \overline{\tau^\dagger_f}. \\ \qquad\quad \texttt{letrec this} = \quad \texttt{let}\ o = \{\overline{f = y}, \overline{m = e^\dagger_m}\}\quad \texttt{in new}(\texttt{fst}(\texttt{unfold}(x)); o) \\ \qquad\quad \texttt{in this}\rangle)\quad \texttt{in}\ e^\dagger\quad : \tau^\dagger\end{array}}$$

Figure 10 Source Language to Core Language Expression Translation (Part 2 of 2).

match it further later on. Implementing hierarchical tags as nested tagging also fails, because then `tagged` C is not a subtype of `tagged` A.

This insight was not obvious to us when we began; our first, failed, translation attempted to use a single construct combining `extract` and `match`. We believe it will be useful both to theorists who will further study the type-theoretic foundations of objects, and to language designers who wish to provide primitives sufficient to encode rich object systems.

5 Implementation

We implemented the core functionality of tags as a contribution to the open source Wyvern programming language [24] developed at Carnegie Mellon University (CMU) in the USA and Victoria University of Wellington in New Zealand, although the implementation of certain supporting features such as dependent function types are not yet complete. The Wyvern interpreter's open source implementation is available [5] and this paper is accompanied by an artifact that was accepted and archived. The syntax of Wyvern's tag support differs slightly from the source-level language presented in Section 2 to better harmonize with the other features of Wyvern, but the underlying concepts remain the same and the implementation is informed by the theory presented here.

[5] `https://github.com/wyvernlang` (our contribution is in the TaggedTypes branch)

```
tagged type Window
  def draw():Str

type WindowMod
  tagged class Win [case of Window]
    class def make():Win = new
    def draw():Str = ""

val basicWindow:WindowMod = new
  tagged class Win [case of Window]
    class def make():Win = new
    def draw():Str = "blank␣window"

def makeBordered(wm: WindowMod):WindowMod = new
  tagged class Win [case of wm.Win]
    class def make():Win = new
    def draw():Str = "bordered␣window"

def makeScrollable(wm: WindowMod):WindowMod = new
  tagged class Win [case of wm.Win]
    class def make():Win = new
    def draw():Str = "scrollable␣window"

def userWantsBorder():Bool = true

val winMod:WindowMod =
  if (userWantsBorder())
  then
      makeBordered(basicWindow)
  else
      basicWindow

val bigWinMod:WindowMod = makeScrollable(winMod)
val smallWin = winMod.Win.make()
val bigWin = bigWinMod.Win.make()

def screenCap(w:Window):Str
  match(w):
    bigWinMod.Win => "big"
    default => "small"

val result = screenCap(bigWin)

result // result == "big"
```

Figure 11 Wyvern Example with Tagged Types.

Bordered Window Example. To understand our implementation of tags in Wyvern, consider the Wyvern version of the bordered window example (Section 2.1), shown in Figure 11. The code is one of the unit tests, and typechecks and executes correctly in the current Wyvern implementation.

Before we implemented tags, Wyvern supported `type` and `class` declarations. In our tags extension, both of them can be declared as `tagged`. If the type or class declaration is nested within a function – e.g, the `tagged` classes declared in the `makeBordered` and `makeScrollable` – then a fresh tag is created every time the function is executed, as in the semantics of the source and core languages we defined.

Wyvern does not have a way of specifying a class type directly, but we can specify the type of an object that has a class nested within it – that object plays the role of a module. For example, `WindowMod` is the type of a module that defines a `Win` class that (transitively) extends `Window`. The Wyvern syntax for specifying a sub-tagging relationship is `case of`, which is analogous to `extends` in Java.

Here is how `makeBordered` defines a mixin: it accepts a `WindowMod` module as an argument and produces another `WindowMod` module, where the class nested in the result has a subtag of the class nested in the argument. The `makeScrollable` function is similar. The `winMod` module decides whether to apply `makeBordered` dynamically based on the result of `userWantsBorder`, demonstrating Wyvern's support for dynamic composition. In this test case, `userWantsBorder` always returns `true`, but Wyvern's interpreter doesn't know that until it executes the function.

The `screenCap` function shows how even if we do not statically know the type of `w`, we can dynamically match against a tagged class defined in any module, such as `bigWinMod`. When calling `screenCap` with `bigWin` as argument, the match succeeds.

Implementation. Wyvern is currently implemented in an interpreter; when run on a piece of source code, the source is parsed, typechecked, and then interpreted. The operation of the typechecker and interpreter roughly follows the semantics outlined in this paper, appropriately adapted to the more practical design of the Wyvern language. For example, each time we create a new object which contains a nested tagged type, we create a new object in the interpreter representing a fresh tag, and we associate it with the nested type. Objects created of that type are then associated with that tag, and the tag can be checked when `match` statements are executed.

The design of Wyvern includes not only extensible tags as described in this paper, but also closed tagged unions. A `tagged` type can be declared with the `[comprises` $T_1, T_2, ...T_n$`]` modifier, which specifies that the type being declared has only the subtags listed in the `comprises` clause. The listed types must declare themselves as a case-of of the parent tag, and the typechecker ensures that no other types are declared to be a case of that parent tag. This allows tagged types to simulate not only objects, but also algebraic datatypes.

6 Modeling Multiple Inheritance

So far, we have developed a foundational type theory that can explain the common constructs of single-inheritance object-oriented languages. OO languages with multiple inheritance, however, cannot be modeled with our calculus so far. A key barrier is subtyping: we want the type `tagged` T to be a subtype of the type `tagged` T_i for each supertag T_i of T, but this cannot be achieved in our current system. This is because in the system presented so far, each tag can have only one parent, and each object can have only one (outermost) tag. Multiple tagging (e.g. by m and n) can be achieved by nesting tags (e.g. a type `tagged` n, where n's inner type is `tagged` m), but this does not provide the desired subtyping properties (in this case, `tagged` n is not a subtype of `tagged` m). The issue is that a tagged object is not identical to its contents. One could imagine trying to make a tagged object semantically identical to its contents, but this seemed both awkward to us and inconsistent with prior type-theoretic models of tags (e.g. in Glew's work and standard ML [13, 14, 16]) and we did not pursue it.

While the restriction to single inheritance is useful to keep the system simple in the main presentation above, it can easily be relaxed. Figure 12 shows the changes to the syntax and static semantics necessary to support OO languages with multiple inheritance. We allow a subtag to extend multiple previously-defined tags; the type being tagged must be a subtype of all of the types tagged by its supertags. When creating an object, it can be tagged with multiple tags as well, and the initial value provided must likewise have a type

$e ::= \texttt{newtag}[\tau] \mid \texttt{subtag}[\tau](n) \mid \texttt{new}(N; e) \mid \texttt{match}(e; N; x.e; e) \mid \texttt{extract}(n; e) \mid \ldots$

$\tau ::= Tag(\tau) \mid \texttt{tagged}\ N \mid \ldots$

$Tag(\tau) ::= \tau\ \texttt{tag} \mid \tau\ \texttt{tag extends}\ n$

Where $N = \{\overline{n}\}$.

$$\frac{\forall n \in N \qquad \Gamma \vdash_\Sigma n : Tag(\tau) \qquad \Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \texttt{new}(N; e) : \texttt{tagged}\ N}(\text{MTag})$$

$$\frac{\Gamma \vdash_\Sigma \texttt{tagged}\ N \subseteq \texttt{tagged}\ N' \qquad \begin{array}{c}\Gamma \vdash_\Sigma e_1 : \texttt{tagged}\ N' \\ \Gamma, x : \texttt{tagged}\ N \vdash_\Sigma e_2 : \tau\end{array} \qquad \Gamma \vdash_\Sigma e_3 : \tau}{\Gamma \vdash_\Sigma \texttt{match}(e_1; N; x.e_2; e_3) : \tau}(\text{MMatch})$$

$$\frac{\Gamma \vdash_\Sigma e : \texttt{tagged}\ N \qquad n \in N \qquad \Gamma \vdash_\Sigma n : Tag(\tau)}{\Gamma \vdash_\Sigma \texttt{extract}(n; e) : \tau}(\text{MExtract})$$

Where $\Gamma \vdash_\Sigma \texttt{tagged}\ N \subseteq \texttt{tagged}\ N'$ iff $\forall n \in N. \exists n' \in N'. \Gamma \vdash_\Sigma \texttt{tagged}\ n \updownarrow \texttt{tagged}\ n'$

Figure 12 Static Semantics for Multiple Tags.

that matches (perhaps via subtyping) the types expected by each tag.[6] Finally, `match` can compare against a set of tags. We leave `extract` to get the contents for one tag, as we would otherwise need to compute an intersection type – something that is feasible but would unnecessarily complicate our presentation. The extensions to the subtyping judgments, value judgments, and dynamic semantics are straightforward; the latter two figures are left to the supplementary material [20] for space reasons.

7 Related Work

First-class classes have been used in Racket along with mixin support; [9] describes the dynamically-typed design, which supports an `implementation?` operation that provides instance (or tag) checking. The topic of first-class classes with static typing has been explored by Takikawa, et al. [27]. They design a gradual typing system to support statically-typed and dynamically-typed portions of their language. They also demonstrate the ability to use mixins via row polymorphism for classes. However, the use of row polymorphism gives their type system a structural flavor; it does not express the concept of an object that is associated with a particular tag.

Reppy and Riecke extend Standard ML with objects and generalize pattern matching to typecase [26]. The extension, called Object ML, adds objects, subtyping, and heterogeneous collections to SML. The design of objects in OML is motivated by recursive types, but opts to keep objects as second-class declarations. Furthermore, they also use SML structs to

[6] We debated between supporting multiple inheritance with tags that have multiple parents vs. objects that have multiple tags. We chose the multiply-tagged object solution as it seemed cleaner and easier to formalize; for example, we can leave the formalization of the tag hierarchy unchanged. Furthermore, the multiply-tagged object design matches the intuition that typical uses of multiple inheritance can be viewed a combination of several dimensions of reuse, each of which can be described in a single inheritance hierarchy [22]. Nevertheless, both choices are possible.

encode classes in OML. As such, their tags are second-class, while our source language opts to use first-class classes and thus translates to first-class tags. Other work on extensible datatypes in functional languages also incorporates second-class tags, thus differing from our results in similar ways [3, 23].

Abadi, et al introduced a means of encoding dynamic types in a statically-typed language through the use of (non-hierarchical) tagging and typecasing [1]. Vytiniotis, et al. explores open and closed type casing through their $\lambda_{\mathcal{L}}$ language [28]. They were primarily concerned with open and closed datatypes and those two different forms of ad-hoc polymorphism. They analyze sets of tags in order to statically check whether or not a given typecase type checks. They do not provide typecasing with subtyping or any form of hierarchical typing.

Our core language is similar to Glew's source language, which was used to motivate a more general use of hierarchical store for modeling type dispatch [13]. However, Glew's language does not use the static type system to keep track of relationships between tags, opting instead to rely on the operational semantics to uphold the theorems of type safety. In Glew's system, for example, the static type of an object is simply `tagged`; it does not track the object's class (or even a superclass), which is a basic capability of OO type systems that we wish to model.

Others have explored tag-like constructs in the context of object-oriented languages. The Unity language defined a *brand* construct that provided nominal types in an otherwise structurally-typed language; at run time, brands are associated with tags that are then used for dispatch [21]. Brands in unity are second-class. Concurrent with our work, Jones *et al.* defined a first-class brand system for Grace [19]. Their focus is on a primitive brand construct that adds nominality to an otherwise structurally-typed system; reflection is used to test brand membership. In contrast, our focus is on the type theory of tags, and thus our work differs in providing primitive operations taken from type theory, including an explicit match operation, and statically tracking the sub-tagging hierarchy.

8 Conclusion

The previous sections of this paper explored a foundational account of class-based object-oriented languages: one that supports dynamic class creation and composition out of mixins, and explains classes in terms of a novel primitive hierarchical tag construct inspired by the type theory of extensible sums. We hope that this account contributes to a better understanding of the relationship between the constructs of typical object-oriented programming languages and the fundamental elements of type theory. Our account highlights that the most simple tag constructs are insufficient to model objects in a type-preserving way, yet shows that small extensions to support static reasoning about tag hierarchy, to provide an appropriate `match` construct, and to statically track the tag associated with each object, are sufficient for modeling objects. We discovered an interesting subtlety, discussed at the end of section 4, in the need to separate `match` from `extract`, and in section 6 we discussed how our approach naturally generalizes to support objects with multiple tags and tags with multiple supertags. These results suggest that functional programming languages, such as Standard ML, that already provide an extensible tag mechanism [16] could gain expressiveness by adopting the enhanced constructs in our theory, perhaps in conjunction with mechanisms such as refinement types [11].

We expect the theory presented here to contribute to the design of Wyvern [24], a language that already supports dynamic class creation and composition, but may benefit from support for multiple tags as well. The flexibility of the theory suggests that it may enable statically

typed, nominal OO languages to express many useful design idioms that, at present, are limited to dynamically-typed or structurally-typed languages.

Acknowledgments. This work was supported by the U.S. National Security Agency lablet contract #H98230-14-C-0140. We acknowledge an invaluable input of Benjamin Chung – the primary maintainer of the Wyvern Compiler.

References

1 Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. *Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

2 Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *European Conference on Object-Oriented Programming*, 1999.

3 François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *Principles of Programming Languages*, 1997.

4 Gilad Bracha and William Cook. Mixin-based inheritance. In *Object-Oriented Programmming, Systems, Languages, and Applications*, 1990.

5 Kim Bruce, Luca Cardelli, and Benjamin Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, 1999.

6 Luca Cardelli. Amber. In *Combinators and Functional Programming Languages*, 1986.

7 David Clarke, James Noble, and John Potter. Simple ownership types for object confinement. In *European Conference on Object-Oriented Programming*, 2001.

8 Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *European Conference on Object-Oriented Programming*, 2007.

9 Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Asian Symposium on Programming Languages and Systems*, 2006.

10 Matthew Flatt, Shriram Krishnamurthi, and Mattias Felleisen. Classes and mixins. In *Principles of Programming Languages*, 1998.

11 Tim Freeman and Frank Pfenning. Refinement types for ML. In *Programming Language Design and Implementation*, June 1991.

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

13 Neal Glew. Typed dispatch for named hierarchical types. In *International Conference on Functional Programming*, 1999.

14 Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2013.

15 Robert Harper and Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages (Edited by Benjamin C. Pierce)*, chapter Design Considerations for ML-Style Module Systems. MIT Press, 2005.

16 Robert Harper and Chris Stone. An interpretation of Standard ML in type theory. Technical Report CMU-CS-97-147, Carnegie Mellon University, 1997.

17 Susumu Hayashi. Singleton, union and intersection types for program extraction. In *Theoretical Aspects of Computer Software*, 1991.

18 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

19 Timothy Jones, Michael Homer, and James Noble. Brand objects for nominal typing. In *European Conference on Object-Oriented Programming*, 2015.

20 Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. A theory of tagged objects (with supplementary material). Technical Report 15-03, ECS, VUW, 2015. `http://ecs.victoria.ac.nz/Main/TechnicalReportSeries`.

21 Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *European Conference on Object-Oriented Programming*, 2008.

22 Donna Malayeri and Jonathan Aldrich. CZ: Multiple inheritance without diamonds. In *Object-Oriented Programmming, Systems, Languages, and Applications*, 2009.

23 Todd Millstein, Colin Bleckner, and Craig Chambers. Modular typechecking for hierarchically extensible datatypes and functions. *Transactions on Programming Languages and Systems*, 26(5):836–889, 2004.

24 Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Wyvern: A simple, typed, and pure object-oriented language. In *Mechanisms for Specialization, Generalization, and Inheritance (MASPEGHI)*, 2002.

25 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

26 John Reppy and Jon Riecke. Simple objects for Standard ML. In *Programming Language Design and Implementation*, 1996.

27 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Mattias Felleisen. Gradual typing for first-class classes. In *Object-Oriented Programmming, Systems, Languages, and Applications*, 2012.

28 Dimitrios Vytiniotis, Geoffrey Washburn, and Stephanie Weirich. An open and shut typecase. In *Programming Language Design and Implementation*, 2005.