Combining Structural Subtyping and External Dispatch

Donna Malayeri Jonathan Aldrich
Carnegie Mellon University
{donna+, aldrich+}@cs.cmu.edu

Abstract

Nominal subtyping (or user-defined subtyping) and structural subtyping each have their own strengths and weaknesses. Nominal subtyping allows programmers to explicitly express design intent, and, when types are associated with run time tags, enables run-time "type" tests (e.g., downcasts) and external/multimethod dispatch. On the other hand, structural subtyping is flexible and compositional, allowing unanticipated reuse. To date, nearly all object-oriented languages fully support only one subtyping paradigm or the other.

In this paper, we describe a core calculus for a language that combines the key aspects of nominal and structural subtyping in a unified framework. Our goal is to combine the flexibility of structural subtyping while still allowing static typechecking of external methods. We prove type safety for this language and illustrate its practical utility through examples that are not easily expressed in other languages. Our work provides a clean foundation for the design of future languages that enjoy the benefits of both nominal and structural subtyping.

1. Introduction

In the research community, structural subtyping is considered a clean and theoretically pleasing account of subtyping. However, the most widely used object-oriented languages provide little or no support for structural subtyping, relying instead on user-defined nominal subtyping. Each kind of subtyping has its merits, but no existing language combines the two in a uniform way.

Nominal subtyping allows the programmer to express and enforce design intent explicitly. A programmer's defined subtyping hierarchy serves as checked documentation that specifies how the various parts of a program are intended to work together. Explicit specification also has the advantage of preventing "accidental" subtyping relationships, such as the standard example of `cowboy.draw()` and `circle.draw()`. Nominal subtyping also allows recursive types to be easily and transparently defined, since recursion can simply go through the declared names. Another advantage is that error messages are usually much more comprehensible, since (for the most part) every type in a type error is one that the programmer has defined explicitly. Finally, and most importantly, nominal subtyping is necessary for run-time subtyping tests (such as downcasts) as well as external and multimethod dispatch.

On the other hand, structural subtyping is often more expressive than nominal subtyping. It is compositional and intrinsic, existing outside of the mind of the programmer. This has the advantage of supporting unanticipated reuse—programmers don't have to plan for all possible reuse scenarios. Additionally, structural subtyping is often more notationally succinct. Programmers can concisely express type requirements without having to define an entire subtyping hierarchy. In nominal systems, situations can arise which require multiple inheritance or an unnecessary proliferation of types; in a structural system, the required subtyping properties just arise naturally from a few simple axioms. Finally, structural subtyping is far superior in contexts where the structure of the data is of primary importance, such as in data persistence or distributed computing. In contrast, nominal subtyping can lead to unnecessary versioning problems: if some class C is modified to C', C' objects cannot be sent to a distributed process with the original definition C, even if C' is a strict extension of C.

We believe that external dispatch and multimethods are essential constructs for modular and extensible object-oriented programming. External dispatch allows methods to be added to existing classes and multimethods permit method dispatch to depend on any subset of arguments to a function. As many have observed (for instance, [6, 7, 18]), in the absence of external dispatch, one must resort to unwieldy double-dispatch code (such as the "Visitor" design pattern [13]), which, aside from being difficult to understand, forces programmers to plan ahead for this kind of extensibility and further makes it difficult to add new subclasses.

Consequently, our aim is to create a language with a uniform object model that retains the expressive and compositional nature of structural subtyping while permitting external and multimethod dispatch. We present a core calculus, in the style of [14, 10, 1, 5], for a language called Unity, which we believe achieves this goal.

In Unity, an object type is a record type tagged with a "brand."[1] Brands induce the "nominal subtyping" relation, which we call "sub-branding." (The name "brand" is borrowed from Strongtalk [3], which in turn borrowed it from Modula-3 [19].) Brands are "nominal" in that the user defines the sub-brand relationship, like the subclass relation in languages like Java, Eiffel, and C++.

When a brand β is defined, the programmer lists the minimum fields that any objects tagged with β must include. In other words, if the user defines the brand `Point` as having the fields `(x : int, y : int)`, then any value tagged with `Point` must include at least the labels `x` and `y` (with `int` type)—but it may also contain additional fields. Subtyping takes into account both the nominal sub-brand relationship and the usual structural subtyping relationship (width and depth) on records.

To integrate these two relationships, brand extension is constrained: the associated record types must be subtypes. In other words, a brand β can be declared as a sub-brand of

[1] Note that record "types" are not actually types in the true sense, as they do not classify any values—they occur only as part of object types.

θ only if β's associated record type is a subtype of θ's record type. As an example, suppose the brand `3DPoint` is defined as `3DPoint(x : int, y : int, z : int)`. `3DPoint` can be declared as a sub-brand of `Point`, since `(x : int, y : int, z : int)` is a sub-record of `(x : int, y : int)`. However, a brand `1DPoint(x : int)` cannot be a sub-brand of `Point` (since it lacks the `y` field), nor can `FloatingPoint(x : float, y : float)` (since `float` is not a subtype of `int`).

Brands allow external dispatch through a `case` construct that performs tag analysis. This construct allows new methods to be written over existing class hierarchies.

To simplify our model and to focus on the core issues, our calculus requires that the entire brand hierarchy be stated at the start of the program. For similar reasons, the calculus also requires that all branches of the `case` expression be given at once; later extensions would require modifying the code directly. We expect that techniques similar to those of EML [18] would allow us to remove these restrictions in further refinements of our language. (Semantically, each branch of the `case` expression can be viewed as a separate branch of an external method; such branches need not be textually adjacent.)

The contributions of this paper are as follows:

- a language design, Unity, that provides user-defined and structural subtyping relationships in a novel and uniform way. Unity combines the flexibility of external dispatch with the conceptual clarity of width and depth subtyping.
- a formalization of the design of Unity, along with proofs of type safety (Section 3).
- examples that illustrate the expressiveness and flexibility of the language (Section 2). We contrast Unity with other languages in Section 2.1.

2. Examples

The following examples introduce Unity and illustrate the flexibility that comes from the combination of structural subtyping and external methods.

2.1 Example 1: a window toolkit

Figure 1 contains a code excerpt for a windowing system and illustrates the novel features of Unity. The brand `Window` is the top-level type; its fields have been abbreviated by γ. By default, a window does not have a scrollbar. The brands `Textbox` and `StaticText` extend `Window`, and also do not scroll by default. The type `Window(`γ`, s: Scrollbar)` represents a window that does have a scrollbar; the `scroll` function applies to any such window. For this example, we assume that the implementation of `scroll` only needs to access the scrollbar field and the fields in γ.

Let us assume that in a non-scrolling textbox the user can only enter a fixed number of characters. Consequently, we define the brand `ScrollingTextbox` in order to change textbox functionality—in particular, the behavior of the `insertChar` function. Note that `ScrollingTextbox(...)` is a subtype of `Window(`γ`, s: Scrollbar)`, so the `scroll` function can be applied to objects of this type without any additional modification.

If other sub-brands of `Window` (such as `StaticText`) do not need to change their existing behavior when a scrollbar is added, no new sub-brands need be defined. Scrolling functionality can be added to these types by including a `Scrollbar` field, and the `scroll` function is then applicable. Since a textbox that scrolls allows the user to enter more text than the window size permits, a new sub-brand had to be defined so that its case in `insertChar` could be overridden.

```
abstract brand Window (γ)

concrete brand Textbox (γ, currentPos : int)
               extends Window
concrete brand StaticText (γ, text : string)
               extends Window
concrete brand ScrollingTextbox
               (γ, currentPos : int, s : Scrollbar)
               extends Textbox


fun scroll ( w : Window(γ, s : Scrollbar) ) : unit =
     ... // code that performs the scrolling operation

fun insertChar (t : Textbox(···), c : Char) : unit =
     case t of
        | Textbox =>
          ...  // insert a character only if it will fit in the window

        | ScrollingTextbox =>
          ...  // insert the character, scrolling if necessary
```

Subtyping relationships

```
Window (γ, s : Scrollbar) ≤ Window (γ)

Textbox (...) ≤ Window (γ)
ScrollingTextbox (...) ≤ Textbox (...)
ScrollingTextbox(...) ≤ Window(γ, s : Scrollbar)

StaticText(...) ≤ Window (γ)
StaticText(..., s : Scrollbar) ≤ Window(γ, s : Scrollbar)
```

Figure 1. Unity code for a windowing system with textboxes, static text, and scrollbars. The metavariable γ denotes additional fields that would be present in an actual code. Nominal subtyping allows the brand `ScrollingTextbox` to change the behavior of `insertChar` through tag dispatch, while structural subtyping allows the `scroll` function to apply to any window with an `s : Scrollbar` field.

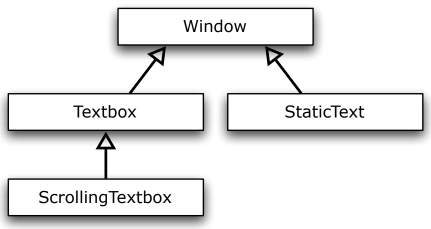
The subtyping relationships induced by these brand definitions are shown below the code listing in Figure 1. Note that, interestingly, `ScrollingTextbox(...)` is a subtype of both `Window(`γ`, s : Scrollbar)` and of `Textbox(...)`, but there is no multiple inheritance involved.

This example illustrates the two kinds of extensibility that Unity provides: structural extension and brand extension.

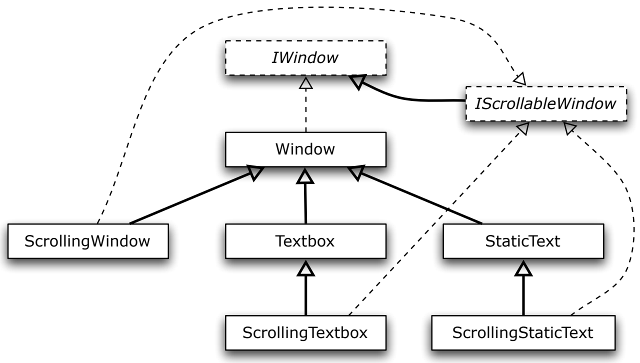
- **Structural extension** can be used to create a new type that adds fields to an existing brand, *without defining new behavior for the resulting type*. So, a `Scrollbar` can be added to a `StaticText` object without defining a new brand, as the existing functionality of the static text box does not need to change if a scrollbar is added.
- **Brand extension** creates a new brand that can be used in dispatch; as a consequence, programs can *define new behavior for the newly defined brand*. Here, `ScrollingTextbox` is defined as an extension of `Textbox` because the behavior of `insertChar` is different depending on whether or not the text box has a scrollbar attached to it.

Comparison to other systems

Java. In Java-like languages, expressing this example would be unwieldy. A common way to express the necessary constraints would involve first defining two interfaces, `IWindow` and



(a) The windowing example as implemented in Unity. Depicted here are the brands that must be defined in order to obtain the desired subtyping relationships.



(b) The same example implemented in Java. Dashed rectangles are interfaces; solid rectangles are classes. Dashed lines indicate the `implements` relationship and solid lines indicate `extends`.

Figure 2. For the windowing example, the user-defined subtyping relationships necessary in Unity vs. those necessary in Java.

`IScrollableWindow`. (Since Java interfaces cannot include fields, the scrollbar field would actually have to be a `getScrollbar()` method in `IScrollableWindow`.) Any window class that wished to allow the possibility of adding a scrollbar, even without changing any other functionality, would have to define a subclass that also implemented `IScrollableWindow`. In this example, we would define the classes `ScrollingWindow`, `ScrollingTextbox`, and `ScrollingStaticText`, though only `ScrollingTextbox` needs to change any functionality; see the class diagram Figure 2(b). Contrast this with the brand structure of the Unity program depicted in Figure 2(a).

Presumably, the Java equivalent of the `scroll` function would be a static function of some helper class and would take an object of type `IScrollableWindow`. Of course, if a programmer defined a new scrolling window class with the correct `getScrollbar()` method, but forgot to implement `IScrollableWindow`, the `scroll` function could not be used on objects of that class. (This situation often arises in Java programs, particularly when one wishes to use library code that is not even aware of the interface that it should be declared to implement.)

There are also some oddities in the Java example. The Java class `ScrollingWindow` is semantically analogous to the Unity type `Window(`γ`, s: Scrollbar)`, but `ScrollingTextbox` and `ScrollingStaticText` are *not* subclasses of `ScrollingWindow`, while the corresponding Unity types *are* subtypes of `Window(`γ`, s: Scrollbar)`. This illustrates the lack of expressiveness that is inherent in languages that require the programmer to name all relevant subtyping relationships.

Traits. A language with traits [23, 24, 20] would provide a much cleaner solution than Java, but would ultimately lack the expressiveness of Unity's structural subtyping. This is due to the fact that traits are mainly designed to solve issues of implementation inheritance (especially multiple inheritance), which is largely orthogonal to the issues we are considering.

Structural subtyping. A language with support for structural subtyping (e.g., O'Caml [16], PolyTOIL [4], etc.) would elegantly express all of the desired subtyping relationships, but these languages allow only internal dispatch—that is, all methods must be defined inside the class definition. In our language, `insertChar` can be an external method; it need not reside inside the definitions of `Textbox` and `ScrollingTextbox`. It would be decidedly non-trivial to add support for external dispatch or multimethods to these types of languages.

Moby. The language MOBY is in many ways similar to Unity, as it supports structural subtyping and a variety of tag subtyping through its inheritance-based subtyping mechanism [11, 12]. This allows expressing many useful subtyping constraints, but MOBY's class types are not integrated with object types in the same way as in Unity. For instance, in MOBY, it is not possible to express the constraint that an object should have a particular class *and* should have some particular fields (that are not defined in the class itself). Additionally, the object-oriented core of MOBY supports only internal dispatch rather than external or multimethod dispatch.

MOBY does include "tagtypes" that are very similar to our brands. These can be used to support downcasts or to encode multimethods, but they are orthogonal to class and object types.

Cecil. Cecil [6] would allow expressing all of the necessary relationships (though new classes do need to be defined for `ScrollingWindow` and `ScrollingStaticText`), as it has a very powerful—but very complex—type system. To write the `scroll` function, a programmer would have to use bounded quantification and a "where" clause constraint, the latter being typechecked via a constraint solver. That is, in words, the argument to `scroll` would have type:

$$\begin{array}{l}\textsf{for all T where}\\ \textsf{T} <: \textsf{Window and signature getScrollbar(T) : Scrollbar}\end{array}$$

Cecil also fully supports multimethod dispatch. Although Unity is strictly less expressive than Cecil, its advantage lies in unifying structural typing with external dispatch in a simpler, more uniform way.

2.2 Example 2 : AST nodes in an IDE

Suppose we have an integrated development environment that includes an editor and a compiler. The top portion of Figure 3 contains an excerpt of a simplified version of the code for such a system. Here, the brands `PlusNode`, `Num` and `Var` define a simple abstract syntax tree. Using structural subtyping, AST nodes with additional information can be created, such as a node with a `loc` field specifying the file location of the code corresponding to the node. Additional functions are available for such nodes, such as the function `highlightNode` that highlights a node's source code in the text editor.

Initial version of the code:

```
abstract brand AstNode() extends Top

concrete brand PlusNode ( n1 : AstNode(), n2 : AstNode() ) extends AstNode
concrete brand Num ( val : int ) extends AstNode
concrete brand Var ( s : Symbol) extends AstNode


// highlight the text corresponding to 'node' in the text editor,
// using the location specified by the 'loc' field
fun highlightNode ( node : AstNode(loc : Location) ) : unit = ...

// compile code to an output stream
fun compile ( node : AstNode(), out : OutStream ) : unit =
    case node of
       | plus as PlusNode => compilePlus(plus, out);
       | num  as Num => ...
       | var  as Var => ...
```

New code for including debug information for some AST nodes:

```
// AST nodes with debug information
concrete brand DebugPlusNode ( n1 : AstNode, n2 : AstNode, loc : Location ) extends PlusNode
concrete brand DebugNum ( val : int, loc : Location ) extends Num
concrete brand DebugVar ( s : Symbol, loc : Location, varName : string ) extends Var

// compile code to an output stream, outputing additional information for Debug* nodes
fun compile ( node : AstNode(), out : OutStream ) : unit =
    case node of
       | plus as PlusNode => compilePlus(plus, out);
       | ...

       // for the debug code, output the file location. for DebugVar, also the variable name
       | dplus as DebugPlusNode => compilePlus(dplus, out); outputLocation(out, dplus.loc)
       | dnum  as DebugNum => ...
       | dvar  as DebugVar => ...
```

Figure 3. Example 2: AST nodes in an IDE. The top portion is the code before changes to add debug information to the AST. The function `highlightNode` makes use of structural information, the external dispatch in `compile` changes its behavior for the declared `Debug*` sub-brands. (We assume here that the newly defined brands have been defined in the preamble of the source program, along with the original brands.)

Note that we did not define a new brand for AST nodes that include file location information. Whether or not a node contains file information, functions that operate over AST nodes do not need to change their behavior, so in this case structural subtyping suffices.

Suppose now that the programmer wishes to add "debug" versions of these AST nodes that contain additional information to be output when compiling in debug mode. For example, a `DebugNum` has a `Location` field, while `DebugVar` includes a `Location` field as well as a string representation of the variable name. The newly added code is listed in the bottom portion of Figure 3.

Since each of these brands have been defined as extensions, they may also customize the behavior of `compile` to output this additional information when compiling. Additionally, since all of the `Debug*` brands have a `Location` field, the function `highlightNode` can be used on objects of this type.

This example again illustrates the expressiveness that is gained by combining nominal and structural subtyping; `highlightNode` makes use of additional structural information, while `compile` relies on tag dispatch to behave differently in different situations.

3. Formal System

The grammar for our language, Unity, is displayed in Figure 4. To define brands, the `brand` construct is used. A brand can be either `abstract` or `concrete`; in the former case, objects tagged with that brand cannot be created, similar to Java's abstract classes. We use the metavariables β and θ to range over brand names.

When a brand is defined, a name is given for it, as well as a list of zero or more (label : type) pairs; these are the labels that must be given values when objects of that brand are created. As previously mentioned, we informally call this list of label-and-type pairs a record "type." The metavariable γ ranges over these record types. The abbreviation $\overline{\ell : \tau}$ denotes $\ell_i \ : \ \tau_i \ ^{i \in 1..n}$ and $\overline{\ell = v}$ denotes $\ell_i = v_i \ ^{i \in 1..n}$.

If β is a brand name, then $\widehat{\beta}$ is the tag value corresponding to β. In other words, $\beta(\gamma)$ is a type, and $\widehat{\beta}$ is its run-time analogue.

As previously stated, the brand subtree must be given in full at the start of the program; this restriction is imposed by the grammar. Note our language currently allows only single inheritance; supporting multiple inheritance is the subject of our future work.

$$
\begin{array}{lll}
\text{Modifiers} & m & ::= \mathsf{abstract} \mid \mathsf{concrete} \\
\text{Programs} & p & ::= e \mid m\ \mathsf{brand}\ \beta(\overline{\ell:\tau})\ \mathsf{extends}\ \beta\ \mathsf{in}\ p \\
\text{Expressions} & e & ::= () \mid x \mid \lambda x{:}\tau.\,e \mid e\,e \\
 & & \quad \mid \widehat{\beta}(\overline{\ell = e}) \mid e.\ell \\
 & & \quad \mid \mathsf{case}\ e\ \mathsf{of}\ \{x_i\ \mathsf{as}\ \widehat{\beta}_i \Rightarrow e_i\ ^{i\in 1..n}\} \\
\text{Types} & \tau & ::= \mathsf{unit} \mid \tau \rightarrow \tau \mid \beta(\overline{\ell:\tau}) \mid \tau \wedge \tau \\
\text{Values} & v & ::= () \mid \widehat{\beta}(\overline{\ell = v}) \mid \lambda x{:}\tau.\,e \\
 & & \\
\text{Contexts} & \Gamma & ::= \varnothing \mid \Gamma, x:\tau \\
 & \Sigma & ::= \varnothing \mid \Sigma, m\ \beta(\overline{\ell:\tau})\ \mathsf{extends}\ \beta \\
 & \Delta & ::= \varnothing \mid \Delta, \widehat{\beta}\ \mathsf{extends}\ \widehat{\beta}
\end{array}
$$

Conventions

$$\widehat{\beta} \equiv \text{ tag value corresponding to } \beta$$

$$\text{fields}_\Sigma\,(\beta) = \gamma \ \text{ if } \ \beta(\gamma)\ \mathsf{extends}\ \beta' \in \Sigma$$

Figure 4. Grammar for Unity

The rest of the language is similar to the standard lambda calculus with records and variants. For ease of presentation, we have not included recursive functions. The expression $\widehat{\beta}(\overline{\ell = e})$ creates an object that is tagged with $\widehat{\beta}$. The expression $e.\ell$ projects the label with name ℓ from the expression e.

The case analysis construct is superficially similar to that of ML-like languages. In each branch, the bound variable x_i represents the scrutinee at a more precise type; the details of typing objects is explained in Section 3.1.

Our language does not directly include iso-recursive types, but many uses of recursive types can be expressed using the brand mechanism. For example, to define streams, we can write:

```
abstract brand Stream extends Top
concrete brand S(hd : int, tail : Stream)
               extends Stream
```

For lists:

```
abstract brand List() extends Top
concrete brand Cons(hd : int, tail : List())
               extends List
concrete brand Nil() extends List
```

Our language includes a limited form of intersection types. Our motivation for including these is to increase the expressiveness of the case statement, which we describe in detail in Section 3.1.2.

Σ is the subtyping context; it stores the user-declared sub-branding relationships. Δ is the corresponding run-time context. Δ contains a strict subset of the information in Σ—it does not contain whether a brand is abstract or concrete, and it does not keep track of the record type $\overline{\ell : \tau}$ associated with each brand. We assume that Σ and Δ each always contain information about the root of the sub-brand tree, which we call Top (similar to `Object` in Java). In other words, we have $\mathsf{concrete\ Top()\ extends\ Top} \in \Sigma$ and $\widehat{\mathsf{Top}}\ \mathsf{extends}\ \widehat{\mathsf{Top}} \in \Delta$.

Our theorems and judgements assume that all contexts are well-formed, defined as follows:

Definition 3.1 (Well-formed context).
The context Σ is *well-formed*, iff the following conditions hold:

1. there is exactly one entry for each brand β.

Intersection on records

$$(\ell_i : \tau_i\ ^{i\in 1..n}, \gamma') \wedge (\ell_i : \tau'_i\ ^{i\in 1..n}, \gamma'') \overset{\text{def}}{=} (\ell_i : (\tau_i \wedge \tau'_i)\ ^{i\in 1..n}, \gamma', \gamma'')$$

where the labels ℓ_i, γ' and γ'' are mutually exclusive.

Definition of *covers*

$$\widehat{\beta}_i\ ^{i\in 1..n}\ \textbf{covers}_\Sigma\ \beta(\gamma_\beta) \overset{\text{def}}{=}$$
$$\forall\theta\,.\ \Sigma \vdash \theta \sqsubseteq \beta \ \text{ and } \ \mathsf{concrete}\ \theta(\gamma_\theta) \in \Sigma \Rightarrow$$
$$\Sigma \vdash \theta(\gamma_\theta), \beta(\gamma_\beta)\ \textbf{disjoint} \text{ or } \exists j \in 1..n.\ \Sigma \vdash \theta \sqsubseteq \beta_j$$

Figure 5. Auxiliary definitions

2. if $\beta(\gamma)\ \mathsf{extends}\ \theta \in \Sigma$, then $\Sigma \vdash \gamma <: \text{fields}_\Sigma\,\theta$.

3.1 Static Semantics

Here we describe the subtyping and typing judgements shown in Figures 6 and 7.

3.1.1 Subtyping rules

Subtyping comprises three parts: the sub-brand judgement ($\sqsubseteq$), the sub-record judgement ($<:$) and the subtype judgement proper ($\leq$). The last is shown in Figure 6, with auxiliary definitions in Figure 5. The first two judgements are not on types, but on the two components of object types, brands and records. The sub-brand judgement is the reflexive, transitive closure of the declared $\mathsf{extends}$ relation. The sub-record judgement is standard depth and width subtyping on records, using the subtype ($\leq$) judgement for depth subtyping. The rules for these judgements appear in Appendix A.2.

The subtype judgement ($\leq$) ties together the sub-brand and sub-record judgements. This is achieved by the rule SUB-NAME: it states that an object type $\beta_1(\gamma_1)$ is a subtype of $\beta_2(\gamma_2)$ when β_1 is a sub-brand of β_2 ($\beta_1 \sqsubseteq \beta_2$) and γ_1 is a sub-record of γ_2 ($\gamma_1 <: \gamma_2$). There are additional conditions that $\beta_1(\gamma_1)$ **ok** and $\beta_2(\gamma_2)$ **ok**, which ensures that these are valid types. The relevant **ok** rule here is:

$$\frac{\Sigma \vdash (\ell_i : \tau_i\ ^{i\in 1..n}) <: \ \text{fields}_\Sigma\,\beta \qquad \Sigma \vdash \tau_i\ \textbf{ok}\ \ {\scriptstyle (i\in 1..n)}}{\Sigma \vdash \beta(\ell_i : \tau_i\ ^{i\in 1..n})\ \textbf{ok}}$$

The full **ok** judgement appears in Appendix A.1.

Our language includes a limited form of intersection types, à la Davies and Pfenning; the rules SUB-$\wedge R$, SUB-$\wedge L_1$ and SUB-$\wedge L_2$ are borrowed from their work [9]. Our language also allows distribution on intersections of object types, via the rules SUB-BRAND-$\wedge L_1$ and SUB-BRAND-$\wedge L_2$. These rules increase the expressiveness of the system. For example, if we assume the definitions of `Point` and `3DPoint` from the introduction, these rules allow us to conclude: `Point(x : int, y : int, c : color)` $\wedge$ `3DPoint(x : int, y : int, z : int)` $\leq$ `3DPoint(x : int, y : int, z : int, c : color)`. Note that these rules make use of the definition of intersection on record types shown in Figure 5.

The remaining subtyping rules are the standard reflexivity, transitivity, and function subtyping rules.

3.1.2 Typing rules

Full typing rules appear in Figure 7. Most of these are standard rules for a simply-typed lambda calculus with subtyping; the novel rules are TP-BRAND-INTRO, TP-NEW-OBJ and TP-CASE.

TP-BRAND-INTRO lets a program introduce new brands as extensions of previous brands, provided that the associated record types are in the appropriate sub-record relationship. The condition $\Sigma \vdash \mathsf{Top}(\gamma)$ **ok** ensures that the record γ is well-formed.

$\boxed{\Sigma \vdash \tau_1 \leq \tau_2}$

$$\frac{}{\Sigma \vdash \tau \leq \tau}\ (\textsc{Sub-Refl}) \qquad \frac{\Sigma \vdash \tau_1 \leq \tau_2 \qquad \Sigma \vdash \tau_2 \leq \tau_3}{\Sigma \vdash \tau_1 \leq \tau_3}\ (\textsc{Sub-Trans}) \qquad \frac{\Sigma \vdash \sigma_1 \leq \tau_1 \qquad \Sigma \vdash \tau_2 \leq \sigma_2}{\Sigma \vdash \tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2}\ (\textsc{Sub-Func})$$

$$\frac{\Sigma \vdash \beta_1 \sqsubseteq \beta_2 \qquad \Sigma \vdash \gamma_1 <: \gamma_2 \qquad \Sigma \vdash \beta_1(\gamma_1)\ \mathbf{ok} \qquad \Sigma \vdash \beta_2(\gamma_2)\ \mathbf{ok}}{\Sigma \vdash \beta_1(\gamma_1) \leq \beta_2(\gamma_2)}\ (\textsc{Sub-Name}) \qquad \frac{\Sigma \vdash \tau \leq \sigma_1 \qquad \Sigma \vdash \tau \leq \sigma_2}{\Sigma \vdash \tau \leq \sigma_1 \wedge \sigma_2}\ (\textsc{Sub-}\wedge R)$$

$$\frac{\Sigma \vdash \tau_1 \leq \sigma}{\Sigma \vdash \tau_1 \wedge \tau_2 \leq \sigma}\ (\textsc{Sub-}\wedge L_1) \qquad \frac{\Sigma \vdash \tau_2 \leq \sigma}{\Sigma \vdash \tau_1 \wedge \tau_2 \leq \sigma}\ (\textsc{Sub-}\wedge L_2) \qquad \frac{\Sigma \vdash \beta_1 \sqsubseteq \beta_2}{\Sigma \vdash \beta_1(\gamma_1) \wedge \beta_2(\gamma_2) \leq \beta_1(\gamma_1 \wedge \gamma_2)}\ (\textsc{Sub-Brand-}\wedge L_1)$$

$$\frac{\Sigma \vdash \beta_2 \sqsubseteq \beta_1}{\Sigma \vdash \beta_1(\gamma_1) \wedge \beta_2(\gamma_2) \leq \beta_2(\gamma_1 \wedge \gamma_2)}\ (\textsc{Sub-Brand-}\wedge L_2)$$

Figure 6. Subtyping rules for Unity

$\boxed{\Gamma \mid \Sigma \vdash p : \tau}$

$$\frac{\beta_1 \notin \Sigma \qquad \Sigma \vdash \mathsf{Top}(\gamma)\ \mathbf{ok} \qquad \Sigma \vdash \gamma\ <:\ \mathsf{fields}_\Sigma\, \beta_2 \qquad \Gamma \mid \Sigma, m\ \beta_1(\gamma)\ \mathsf{extends}\ \beta_2 \vdash p : \tau' \qquad \Sigma \vdash \tau'\ \mathbf{ok}}{\Gamma \mid \Sigma \vdash m\ \mathsf{brand}\ \beta_1(\gamma)\ \mathsf{extends}\ \beta_2\ \mathsf{in}\ p : \tau'}\ (\textsc{Tp-Brand-Intro})$$

$\boxed{\Gamma \mid \Sigma \vdash e : \tau}$

$$\frac{x : \tau \in \Gamma}{\Gamma \mid \Sigma \vdash x : \tau}\ (\textsc{Tp-Var}) \qquad \frac{}{\Gamma \vdash ()\ :\ \mathsf{unit}}\ (\textsc{Tp-Unit}) \qquad \frac{\Gamma, x : \tau_1 \mid \Sigma \vdash e : \tau_2}{\Gamma \mid \Sigma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2}\ (\textsc{Tp-Fun})$$

$$\frac{\Gamma \mid \Sigma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \mid \Sigma \vdash e_2 : \tau_1}{\Gamma \mid \Sigma \vdash e_1\, e_2 : \tau_2}\ (\textsc{Tp-App}) \qquad \frac{\Gamma \mid \Sigma \vdash e : \sigma \qquad \Sigma \vdash \sigma \leq \tau}{\Gamma \mid \Sigma \vdash e : \tau}\ (\textsc{Tp-Subs})$$

$$\frac{\mathsf{concrete}\ \beta(\gamma) \in \Sigma \qquad \Sigma \vdash (\ell_i : \tau_i)^{\ i \in 1..n} <: \gamma \qquad \Gamma \mid \Sigma \vdash e_i : \tau_i\ \ (i \in 1..n)}{\Gamma \mid \Sigma \vdash \widehat{\beta}(\ell_i = e_i^{\ i \in 1..n}) : \beta(\ell_i : \tau_i^{\ i \in 1..n})}\ (\textsc{Tp-Brand-Cons})$$

$$\frac{\Gamma \mid \Sigma \vdash e : \beta(\ell_i : \tau_i^{\ i \in 1..n})}{\Gamma \mid \Sigma \vdash e.\ell_k : \tau_k}\ (\textsc{Tp-Proj})$$

$$\frac{\begin{array}{c}\Gamma \mid \Sigma \vdash e : \beta(\gamma) \qquad \{\widehat{\beta}_k^{\ k \in 1..n}\}\ \mathbf{distinct} \\ \Sigma \vdash \beta_i \sqsubseteq \beta\ \ (i \in 1..n) \qquad \Sigma \vdash \widehat{\beta}_k^{\ k \in 1..n}\ \mathbf{covers}\ \beta(\gamma) \qquad \Gamma,\ x_i : \beta_i(\gamma\ \wedge\ \mathsf{fields}_\Sigma\, \beta_i) \mid \Sigma \vdash e_i : \tau\ \ (i \in 1..n)\end{array}}{\Gamma \mid \Sigma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{x_j\ \mathsf{as}\ \widehat{\beta}_j \Rightarrow e_j^{\ \ j \in 1..n}\} : \tau}\ (\textsc{Tp-Case})$$

Figure 7. Typing rules for Unity's programs p and expressions e. Auxiliary definitions appear in Figure 5. Note: a program consisting entirely of an expression is typechecked using the $\Gamma \mid \Sigma \vdash e : \tau$ judgement.

The rule Tp-New-Obj ensures that values of object type that are tagged $\widehat{\beta}$ meet the requirements of the brand β as defined in the define brand construct. Only brands that are concrete may be used to create object values.

The Tp-Case rule illustrates the novel aspects of our system. It is similar in syntax to the "case" constructs of ML-like languages, but its function is to perform dispatch on brand tags. In contrast to ML, for instance, the order in which branches appear does not affect execution; the most specific branch is always selected. The Tp-Case rule ensures that such a branch will always exist.

The first three premises of the rule are fairly simple; brand tags may not be repeated, and each must be a sub-brand of the scrutinee's brand. The **covers** condition ensures that there is a branch for every concrete sub-brand of β. Since the brand hierarchy is fixed at the start of the program, when the case statement is typechecked, all possible sub-brands are known.

The **covers** condition also allows the programmer to omit impossible branches. These can occur when the record type of the scrutinee is incompatible with the record type of a concrete sub-brand of β. It may not be immediately obvious why this might occur, given our restrictions on defining brands and creating objects. But this situation can arise when structural subtyping is used. For example, suppose our system included the base types `int` and `color`, and that these types are disjoint from one another (that is, there are no values that have both type `int` and `color`). Suppose also that the scrutinee was a `Point` that for some reason had a `z` field of type `color`; i.e., `Point(x : int, y : int, z : color)`. In such a case, the programmer may leave omit the branch for `3DPoint`—even though `3DPoint` $\sqsubseteq$ `Point`—since the scrutinee's structural type is incompatible with the record type of `3DPoint`.

Formally, **covers** achieves this effect by making use of the **disjoint** judgement. This judgement, presented in full in Appendix A.1, has the following property:

Lemma 3.1 (Disjoint types).
If $\Sigma \vdash \tau_1, \tau_2$ **disjoint** and $\varnothing \mid \Sigma \vdash v : \tau_1$, then $\varnothing \mid \Sigma \nvdash v : \tau_2$.

In the example from above, we have $\Sigma \vdash$ `Point(x : int, y : int, z : color)`, `3DPoint(x : int, y : int, z : color)` **disjoint**.

The final premise of TP-CASE is probably the most interesting. For simplicity, let us first ignore the intersection that appears in the binding added to Γ so that it reads:

$$\Gamma,\, x_i : \beta_i(\text{fields}_\Sigma\, \beta_i) \mid \Sigma \vdash e_i : \tau \quad (\text{for } i \in 1..n).$$

So, in each branch, the scrutinee is bound to a variable x_i with a more precise type. In particular, it will have the tag $\widehat{\beta}_i$ and the fields defined for β_i in Σ. But, this rule would not allow the programmer to make use of the additional structural information that may be known about the scrutinee—it has the fields γ, which (a) may contain more fields than fields$_\Sigma\, \beta$, or (b) may refine one or more of the types in fields$_\Sigma\, \beta$.

Consequently, to retain as much typing information as possible, the type of x_i is instead $\beta_i(\gamma \wedge \text{fields}_\Sigma\, \beta_i)$. This uses the definition of intersection of records as defined in Figure 5 (distinct labels are concatenated; for common labels, the intersection of the associated types is taken).

For example, assuming the definitions of `Point` and `3DPoint` from above, consider the following program:

```
λe : Point(x : int, y : int, c : color).
  case e of
       p1 as Point => ...
       p3 as 3DPoint => bar(p3)
```

The variable `p3` has type `3DPoint(x : int, y : int, z : int, c : color)`. It may initially appear that this does not lead to a substantial increase in expressiveness, since the fields of the scrutinee are in scope; in this example, `p3.c` is equivalent to `e.c`. However, suppose that the argument to `bar` must have type `3DPoint(..., c : color)`. If the type of `p3` did not include the color field, a new `3DPoint` object could be created with the integer fields of `p3` and the color field of `e`, but this could change the behavior of the program. Based on the semantics of the language, we know that tag(`p3`) $\sqsubseteq$ tag(`3DPoint`); if it is a strict sub-brand (e.g. `3DSubPoint`), then this information would be lost if a new object were created (i.e., the program would have to create a `3DSubPoint` object, rather than a `3DPoint`).

Therefore, including this intersection type in the premise of the case rule strictly increases the expressiveness of the language; we expect that it will be even more relevant in a language extension that includes a form of row polymorphism [21].

3.2 Dynamic Semantics

The full evaluation rules appear in Appendix A. Most are standard; the interesting ones are highlighted in Figure 8.

E-BRAND-DECL adds information from the brand declaration to the run-time subtyping context Δ. Note that the information about the brand's associated record γ is not included, since structural subtyping relationships have no effect on a program's execution. Sub-branding relationships do need to be retained since they affect the dispatch in the evaluation of the case expression.

The E-CASE2 rule handles evaluation of case expressions by appealing to the auxiliary definition of the partial function **select**$_\Delta$. This function returns the index of the most specific matching tag, if one exists. In words, **select**$_\Delta(\widehat{\theta}, \widehat{\beta}_i\ ^{i\in 1..n})$ returns the index j of a $\widehat{\beta}_j$ that is the least upper bound (i.e., the smallest super-brand) of

$$\textbf{select}_\Delta(\widehat{\theta}, \widehat{\beta}_i\ ^{i\in 1..n}) \overset{\text{def}}{=} j \in 1..n,$$
$$\text{where } \Delta \vdash \widehat{\theta} \sqsubseteq \widehat{\beta}_j$$
$$\text{and } \forall k \in 1..n\,.\, \Delta \vdash \widehat{\theta} \sqsubseteq \widehat{\beta}_k \Rightarrow \Delta \vdash \widehat{\beta}_j \sqsubseteq \widehat{\beta}_k$$

$$\frac{}{\begin{array}{l} m \ \mathsf{brand}\ \beta_1(\gamma)\ \mathsf{extends}\ \beta_2\ \mathsf{in}\ p \mid \Delta \\ \quad \longmapsto\ p \mid \Delta, (\widehat{\beta}_1\ \mathsf{extends}\ \widehat{\beta}_2)\end{array}} \ (\text{E-BRAND-DECL})$$

$$\frac{\textbf{select}_\Delta(\widehat{\theta}, \widehat{\beta}_i\ ^{i\in 1..n}) = k}{\begin{array}{l}\mathsf{case}\ \widehat{\theta}(\ell = v)\ \mathsf{of}\ \{x_i\ \mathsf{as}\ \widehat{\beta}_i \Rightarrow e_i\ ^{i\in 1..n}\} \\ \quad \longmapsto_\Delta\ [\,\widehat{\theta}(\overline{\ell = v})\,/x_k\,]\, e_k\end{array}} \ (\text{E-CASE2})$$

Figure 8. Selected evaluation rules

$\widehat{\theta}$. The proof of the progress theorem for this case shows that for well-typed programs, **select**$_\Delta$ is always uniquely defined.

3.3 Type safety

In this section, we describe the key lemmas and interesting cases needed to prove type safety via the standard theorems of progress and preservation.

Due to the fact that our language includes both intersection types and subtyping, the usual lemmas have to be stated very carefully. For example, the typing inversion lemma is as follows:

Lemma 3.2 (Inversion of the typing judgement).

1. If $\Gamma \mid \Sigma \vdash \lambda x{:}\tau_1.\, e : \sigma$ and $\Sigma \vdash \sigma \leq \sigma_1 \rightarrow \sigma_2$ then $\Sigma \vdash \sigma_1 \leq \tau_1$ and $\Gamma, x : \tau_1 \mid \Sigma \vdash e : \sigma_2$.
2. If $\Gamma \mid \Sigma \vdash \widehat{\theta}(\ell_i = e_i\ ^{i\in 1..n}) : \sigma$ and $\Sigma \vdash \sigma \leq \beta(k_j : \tau_j\ ^{j\in 1..m})$ then for some $\sigma_i\ ^{i\in 1..n}$ we have:
   (a) $\Gamma \mid \Sigma \vdash \widehat{\theta}(\ell_i = e_i\ ^{i\in 1..n}) : \theta(\ell_i : \sigma_i\ ^{i\in 1..n})$ and $\Gamma \mid \Sigma \vdash e_i : \sigma_i$
   (b) $\Sigma \vdash (\ell_i : \sigma_i\ ^{i\in 1..n}) <: \text{fields}_\Sigma\, \theta$
   (c) $\Sigma \vdash \theta \sqsubseteq \beta$
   (d) $\Sigma \vdash (\ell_i : \sigma_i\ ^{i\in 1..n}) <: (k_j : \tau_j\ ^{j\in 1..m})$

Note that the lemma includes premises such as $e : \sigma$ and $\sigma \leq \sigma_1 \rightarrow \sigma_2$, rather than the simpler $e : \sigma_1 \rightarrow \sigma_2$; this is due to the presence of intersection types. (Since the expression could have type e.g., $\sigma_1 \rightarrow \sigma_2 \wedge \sigma_1' \rightarrow \sigma_2'$, stating the lemma in terms of subtypes simplifies its proof, as this type is a subtype of, for instance, $\sigma_1 \rightarrow \sigma_2$.)

Similarly, one of the cases in the canonical forms lemma is stated as:

If $\varnothing \mid \Sigma \vdash v : \sigma$ and $\Sigma \vdash \sigma \leq \beta(\gamma)$, v is of the form $\widehat{\theta}(\ell_i = v_i)$.

It would be very difficult to prove a more "standard" canonical forms lemma, since objects need not have types of the form $\beta(\gamma)$; their types may contain one or more intersections (i.e. $\beta_1(\gamma_1) \wedge \beta_2(\gamma_2) \wedge \cdots$).

The type safety theorems make use of a consistency requirement between the static subtyping context and the run-time subtyping context.

Definition 3.2 (Models relation on contexts).
The context Σ models Δ, written $\Sigma \vdash \Delta$, if $|\Sigma| = |\Delta|$ and for every

$$m\ \beta_1(\gamma_1)\ \mathsf{extends}\ \beta_2 \in \Sigma$$

we have

$$\widehat{\beta}_1\ \mathsf{extends}\ \widehat{\beta}_2 \in \Delta.$$

The progress theorem not only requires that the program be well-typed under a subtyping context Σ, but also that it be evaluated under run-time context Δ consistent with Σ, *i.e.*, one such that $\Sigma \vdash \Delta$:

Theorem 3.1 (Progress [programs]). If $\varnothing \mid \Sigma \vdash p : \tau$, for some τ and Σ, then one of the following cases holds:

1. p is a value
2. for Δ such that $\Sigma \vdash \Delta$, there exist p' and Δ' such that $p \mid \Delta \longmapsto p' \mid \Delta'$.

The bulk of the proof follows standard techniques. The most interesting case follows from the following lemma:

Lemma 3.3 (Case statement exhaustiveness).

If $\varnothing \mid \Sigma \vdash \widehat{\theta}(\ell_j = v_j\ ^{j \in 1..m}) : \beta(\gamma)$ and $\Sigma \vdash \widehat{\beta}_i\ ^{i \in 1..n}$ **covers** $\beta(\gamma)$ and $\Sigma \vdash \Delta$, then **select**$_\Delta(\widehat{\theta}, \widehat{\beta}_i\ ^{i \in 1..n})$ is defined.

Proof. (Sketch). By typing inversion, we know that θ is concrete and that $\widehat{\theta}(\overline{\ell = v}) : \theta(\overline{\ell : \tau})$, for some $\overline{\tau}$. From the definition of **covers**, either $\theta(\overline{\ell : \tau}), \beta(\gamma)$ **disjoint**, or there is at least one matching branch for θ. The former cannot hold since that would mean that one value has two types that are **disjoint**. So it suffices to show that there is exactly one matching branch. However, the only way that two matching branches could occur would be if there were multiple inheritance, which is disallowed by the grammar. □

As with progress, the preservation theorem must account for the run-time subtyping context Δ.

Theorem 3.2 (Preservation [programs]).
If $\Gamma \mid \Sigma \vdash p : \tau$ and $\Sigma \vdash \Delta$ and $p \mid \Delta \longmapsto p' \mid \Delta'$, then there exists a Σ' such that $\Sigma' \vdash \Delta'$ where $\Gamma \mid \Sigma' \vdash p' : \tau$.

The proof proceeds by induction on typing derivations and appeals to the standard substitution lemma. (Due to the presence of subtyping and intersection types, it is very difficult to prove this theorem by the more usual induction on the one-step evaluation derivation.) The interesting case is the E-CASE2 subcase of TP-CASE. Here, we know

$$\Gamma, x_k : \beta_k(\gamma \wedge \text{fields}_\Sigma\, \beta_k) \mid \Sigma \vdash e_k : \tau$$

and

$$\widehat{\theta}(\overline{\ell = v}) : \beta(\gamma).$$

It suffices to show $\widehat{\theta}(\overline{\ell = v}) : \beta_k(\gamma \wedge \text{fields}_\Sigma\, \beta_k)$; the result then follows from the substitution lemma. To show this, we make use of the following lemma:

Lemma 3.4. If $\Sigma \vdash \theta \sqsubseteq \beta$ then $\Sigma \vdash \text{fields}_\Sigma\, \theta <: \text{fields}_\Sigma\, \beta$.

This formally proves that sub-branding implies structural subtyping on the associated record types.

The remainder of the preservation proof is straightforward. Full proofs of both theorems (and auxiliary lemmas) are provided in the companion technical report [17].

4. Related Work

The languages that are the most similar to Unity are Cecil [6] and Moby [11]; we compared Unity to these in Section 2. Cecil offers greater expressive power over Unity, but at the cost of a much more complex type system that includes constraint solving. Moby includes structural subtyping and an inheritance-based subtyping that is similar to our sub-branding, but as its primary goal is to combine modules and objects, it does not arrive at the same combination of features as Unity does.

MultiJava [7, 8] and EML [18] both include support for modular typechecking of external methods and multimethods; previous work had performed a whole-program analysis. We intend to build on the EML's techniques when we extend Unity to allow brand definitions to occur anywhere in the program. Lee and Chambers [15] have extended EML to include support for parameterized modules.

Strongtalk [3] presents a structural type system for Smalltalk and also supports named subtyping relationships (from which we borrowed the name "brand"), but these are not combined in the type system. Additionally, since it is a type system for Smalltalk, it supports only the single dispatch model.

Cω's [2] main subtyping model is nominal, though it does support depth subtyping—but not width subtyping—for "anonymous structs," which are similar to our records. Following C#, Cω also has a single dispatch model.

OML [22] is similar to our work in that it has declared subtyping (via `objtype`) along with structural subtyping. However, there are a number of notable differences. In particular, OML wished to retain ML-style type inference and therefore places constraints on the subtyping relation. Additionally, depth subtyping is not permitted during `objtype` extension, and there is no support allow for modular extensibility of methods.

5. Summary and Future Work

We have presented a core calculus that combines nominal and structural subtyping and provided proofs of type safety. We have also described the utility of our system through a series of examples.

There are a number of extensions we would like to make. As previously mentioned, we wish to allow brand extension to occur anywhere in the program, and new branches to be added to external methods without modifying existing code.

We also plan on adding parametric polymorphism and row polymorphism [21]. The latter would greatly add to the expressiveness of our system. First, it would increase the utility of the intersection in the types of bound variables in the branches of the case typing rule; the additional information contained in a row variable could then be retained in case branches. In addition, it would allow for more expressive recursive types. Recall that in the `List` example from Section 3, the definition of `Cons` was:

```
concrete brand Cons(hd : int, tail : List())
                extends List
```

In our current language, it would not be possible to describe a list where every `Cons` element also contains, for example, a `color` field, without defining new brands for this purpose. This is because the type of `tail` above is simply `List()`, and there is no way of specifying that for a particular `Cons` object, its `tail` has the same "extra fields" that it does. Row polymorphism would allow the programmer to express this constraint without having to define brands such as `ColorList`, `ColorCons` and `ColorNil`.

Acknowledgments

We wish to thank Bob Harper for his helpful discussions on our system and William Lovas for his comments on an earlier version of this paper.

References

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[2] Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in Cω. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, 2005.

[3] Gilad Bracha and David Griswold. Strongtalk: typechecking smalltalk in a production environment. In *OOPSLA '93: Proceedings*

of the eighth annual conference on Object-oriented programming systems, languages, and applications, pages 215–230, New York, NY, USA, 1993. ACM Press.

[4] Kim B. Bruce, Angela Schuett, Robert van Gent, and Adrian Fiech. Polytoil: A type-safe polymorphic object-oriented language. *ACM Trans. Program. Lang. Syst.*, 25(2):225–290, 2003.

[5] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Inf. Comput.*, 117(1):115–135, 1995.

[6] Craig Chambers and the Cecil Group. The Cecil language: specification and rationale, version 3.2. Available at http://www.cs.washington.edu/research/projects/cecil/, February 2004.

[7] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–145, New York, NY, USA, 2000. ACM Press.

[8] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. Multijava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst.*, 28(3):517–575, 2006.

[9] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 198–208, New York, NY, USA, 2000. ACM Press.

[10] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. Computing*, 1:3–37, 1994.

[11] Kathleen Fisher and John Reppy. The design of a class mechanism for moby. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 37–49, New York, NY, USA, 1999. ACM Press.

[12] Kathleen Fisher and John Reppy. Inheritance-based subtyping. *Inf. Comput.*, 177(1):28–55, 2002.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[14] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherwieght Java: a Minimal Core Calculus for Java and GJ. In *Object-Oriented Programming Systems, Languages, and Applications*, November 1999.

[15] Keunwoo Lee and Craig Chambers. Parameterized modules for classes and extensible functions. In *ECOOP '06 : Proceedings of the 20th European Conference on Object-Oriented Programming*, 2006.

[16] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system, release 3.09. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`, 2004.

[17] Donna Malayeri and Jonathan Aldrich. Combining structural subtyping and external dispatch. Technical Report CMU-CS-06-178, School of Computer Science, Carnegie Mellon University, December 2006.

[18] Todd Millstein, Colin Bleckner, and Craig Chambers. Modular typechecking for hierarchically extensible datatypes and functions. *ACM Trans. Program. Lang. Syst.*, 26(5):836–889, 2004.

[19] Greg Nelson, editor. *Systems programming with Modula-3*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[20] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stephane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenge. The Scala language specification version 1.0. Technical report, Programming Methods Laboratory, EPFL, Switzerland, 2004.

[21] Didier Rémy. Type checking records and variants in a natural extension of ML. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1989.

[22] John Reppy and Jon Riecke. Simple objects for standard ml. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 171–180, New York, NY, USA, 1996. ACM Press.

[23] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *European Conference on Object-Oriented Programming (ECOOP)*, 2003.

[24] Charles Smith and Sophia Drossopoulou. Chai: Traits for Java-like languages. In *ECOOP '05 : Proceedings of the 19th European Conference on Object-Oriented Programming*, 2005.

A. Formal System

A.1 Typing

Definition A.1 (Covering definition).

$$\widehat{\beta}_i^{\;i\in 1..n}\ \mathbf{covers}_\Sigma\ \beta(\gamma_\beta)\ \overset{\text{def}}{=}$$
$$\forall\theta\,.\ \Sigma\vdash\theta\sqsubseteq\beta\ \text{ and }\ \text{concrete}\ \theta(\gamma_\theta)\in\Sigma\Rightarrow$$
$$\Sigma\vdash\theta(\gamma_\theta),\beta(\gamma_\beta)\ \mathbf{disjoint}\ \text{or}\ \exists j\in 1..n.\,\Sigma\vdash\theta\sqsubseteq\beta_j$$

Definition A.2 (Intersection on γ).

$$(\ell_i:\tau_i^{\;i\in 1..n},\gamma')\ \wedge\ (\ell_i:\tau_i'^{\;i\in 1..n},\gamma'')\ \overset{\text{def}}{=}$$
$$\big(\ell_i:(\tau_i\wedge\tau_i')^{\;i\in 1..n},\gamma',\gamma''\big)$$

where the labels ℓ_i, γ' and γ'' are mutually exclusive.

Well-formed judgement

$\boxed{\tau\ \mathbf{ok}}$

$$\frac{}{\Sigma\vdash\mathsf{unit}\ \mathbf{ok}}\qquad\frac{\Sigma\vdash\tau_1\ \mathbf{ok}\qquad\Sigma\vdash\tau_2\ \mathbf{ok}}{\Sigma\vdash\tau_1\to\tau_2\ \mathbf{ok}}$$

$$\frac{\Sigma\vdash(\ell_i:\tau_i^{\;i\in 1..n})<:\ \mathsf{fields}_\Sigma\,\beta\qquad\Sigma\vdash\tau_i\ \mathbf{ok}\ \ (i\in 1..n)}{\Sigma\vdash\beta(\ell_i:\tau_i^{\;i\in 1..n})\ \mathbf{ok}}$$

Disjoint judgement

$\boxed{\tau_1,\tau_2\ \ \mathbf{disjoint}}$

$$\frac{}{\Sigma\vdash\mathsf{unit},\tau_1\to\tau_2\ \mathbf{disjoint}}\ \textsc{Dis-Unit1}$$

$$\frac{}{\Sigma\vdash\mathsf{unit},\beta(\gamma)\ \mathbf{disjoint}}\ \textsc{Dis-Unit2}$$

$$\frac{}{\Sigma\vdash\tau_1\to\tau_2,\beta(\gamma)\ \mathbf{disjoint}}\ \textsc{Dis-Arrow}$$

$$\frac{\Sigma\vdash\tau_1,\tau_2\ \mathbf{disjoint}}{\Sigma\vdash\tau_1,\tau_1\wedge\tau_2\ \mathbf{disjoint}}\ \textsc{Dis-}\wedge R_1$$

$$\frac{\Sigma\vdash\tau_1,\tau_2\ \mathbf{disjoint}}{\Sigma\vdash\tau_1,\tau_2\wedge\tau_1\ \mathbf{disjoint}}\ \textsc{Dis-}\wedge R_2$$

$$\frac{\Sigma\vdash\tau,\tau'\ \mathbf{disjoint}}{\Sigma\vdash\beta_1(\ldots,\ell:\tau,\ldots),\beta_2(\ldots,\ell:\tau',\ldots)\ \mathbf{disjoint}}\ \textsc{Dis-Rec1}$$

$$\frac{\Sigma\vdash\beta_1\not\sqsubseteq\beta_2\qquad\Sigma\vdash\beta_2\not\sqsubseteq\beta_1}{\Sigma\vdash\beta_1(\gamma_1),\beta_2(\gamma_2)\ \mathbf{disjoint}}\ \textsc{Dis-Rec2}$$

$$\frac{\Sigma\vdash\tau_1,\tau_2\ \mathbf{disjoint}}{\Sigma\vdash\tau_2,\tau_1\ \mathbf{disjoint}}\ \textsc{Dis-Sym}$$

Typing rules

$\boxed{\Gamma\mid\Sigma\vdash p:\tau}$

$$\frac{\begin{array}{c}\beta_1\notin\Sigma\\ \Sigma\vdash\mathsf{Top}(\gamma)\ \mathbf{ok}\qquad\Sigma\vdash\gamma\ <:\ \mathsf{fields}_\Sigma\,\beta_2\\ \Gamma\mid\Sigma,m\ \beta_1(\gamma)\ \mathsf{extends}\ \beta_2\vdash p:\tau'\\ \Sigma\vdash\tau'\ \mathbf{ok}\end{array}}{\Gamma\mid\Sigma\vdash m\ \mathsf{brand}\ \beta_1(\gamma)\ \mathsf{extends}\ \beta_2\ \mathsf{in}\ p:\tau'}\ (\textsc{Tp-Brand-Intro})$$

$\boxed{\Gamma\mid\Sigma\vdash e:\tau}$

$$\frac{x:\tau\in\Gamma}{\Gamma\mid\Sigma\vdash x:\tau}\ (\textsc{Tp-Var})\qquad\frac{}{\Gamma\vdash()\ :\ \mathsf{unit}}\ (\textsc{Tp-Unit})$$

$$\frac{\Gamma,x:\tau_1\mid\Sigma\vdash e:\tau_2}{\Gamma\mid\Sigma\vdash\lambda x{:}\tau_1.\,e:\tau_1\to\tau_2}\ (\textsc{Tp-Fun})$$

$$\frac{\Gamma\mid\Sigma\vdash e_1:\tau_1\to\tau_2\qquad\Gamma\mid\Sigma\vdash e_2:\tau_1}{\Gamma\mid\Sigma\vdash e_1\,e_2:\tau_2}\ (\textsc{Tp-App})$$

$$\frac{\Gamma\mid\Sigma\vdash e:\sigma\qquad\Sigma\vdash\sigma\le\tau}{\Gamma\mid\Sigma\vdash e:\tau}\ (\textsc{Tp-Subs})$$

$$\frac{\begin{array}{c}\text{concrete}\ \beta(\gamma)\in\Sigma\qquad\Sigma\vdash(\ell_i:\tau_i)^{\;i\in 1..n}<:\gamma\\ \Gamma\mid\Sigma\vdash e_i:\tau_i\ \ (i\in 1..n)\end{array}}{\Gamma\mid\Sigma\vdash\widehat{\beta}(\ell_i=e_i^{\;i\in 1..n}):\beta(\ell_i:\tau_i^{\;i\in 1..n})}\ (\textsc{Tp-New-Obj})$$

$$\frac{\Gamma\mid\Sigma\vdash e:\beta(\ell_i:\tau_i^{\;i\in 1..n})}{\Gamma\mid\Sigma\vdash e.\ell_k:\tau_k}\ (\textsc{Tp-Proj})$$

$$\frac{\begin{array}{c}\Gamma\mid\Sigma\vdash e:\beta(\gamma)\qquad\{\widehat{\beta}_k^{\;k\in 1..n}\}\ \mathbf{distinct}\\ \Sigma\vdash\beta_i\sqsubseteq\beta\ \ (i\in 1..n)\qquad\Sigma\vdash\widehat{\beta}_k^{\;k\in 1..n}\ \mathbf{covers}\ \beta(\gamma)\\ \Gamma,\,x_i:\beta_i(\gamma\ \wedge\ \mathsf{fields}_\Sigma\,\beta_i)\mid\Sigma\vdash e_i:\tau\ \ (i\in 1..n)\end{array}}{\Gamma\mid\Sigma\vdash\mathsf{case}\ e\ \mathsf{of}\ \{x_j\ \mathsf{as}\ \widehat{\beta}_j\Rightarrow e_j^{\;j\in 1..n}\}:\tau}\ (\textsc{Tp-Case})$$

A.2 Subtyping

A.2.1 Sub-brand judgement

$\boxed{\Sigma\vdash\beta_1\sqsubseteq\beta_2}$

$$\frac{m\ \beta_1(\gamma)\ \mathsf{extends}\ \beta_2\in\Sigma}{\Sigma\vdash\beta_1\sqsubseteq\beta_2}\ (\textsc{Sub-Brand-Decl})$$

$$\frac{}{\Sigma\vdash\beta\sqsubseteq\beta}\ (\textsc{Sub-Brand-Refl})$$

$$\frac{\Sigma\vdash\beta_1\sqsubseteq\beta_2\qquad\Sigma\vdash\beta_2\sqsubseteq\beta_3}{\Sigma\vdash\beta_1\sqsubseteq\beta_3}\ (\textsc{Sub-Brand-Trans})$$

A.2.2 Sub-record judgement

$\boxed{\Sigma\vdash\gamma_1<:\gamma_2}$

$$\frac{}{\Sigma\vdash\gamma<:\gamma}\ (\textsc{Sub-rec-refl})$$

$$\frac{\Sigma\vdash\gamma_1<:\gamma_2\qquad\Sigma\vdash\gamma_2<:\gamma_3}{\Sigma\vdash\gamma_1<:\gamma_3}\ (\textsc{Sub-rec-trans})$$

$$\frac{\ell_i:\tau_i^{\;i\in 1..n}\ \text{is a permutation of}\ \ell_j:\tau_j^{\;j\in 1..n}}{\Sigma\vdash\ell_i:\tau_i^{\;i\in 1..n}<:\ell_j:\tau_j^{\;j\in 1..n}}\ (\textsc{Sub-rec-perm})$$

$$\frac{}{\Sigma\vdash\ell_i:\tau_i^{\;i\in 1..n}\ <:\ \ell_j:\tau_j^{\;j\in 1..m},n>m}\ (\textsc{Sub-rec-width})$$

$$\frac{\text{for each}\ i,\ \Sigma\vdash\sigma_i\le\tau_i}{\Sigma\vdash\ell_i:\sigma_i^{\;i\in 1..n}\ <:\ \ell_i:\tau_i^{\;i\in 1..n}}\ (\textsc{Sub-rec-depth})$$

A.2.3 Subtype judgement

$\boxed{\Sigma \vdash \tau_1 \leq \tau_2}$

$$\frac{}{\Sigma \vdash \tau \leq \tau}\ \textsc{Sub-Refl}$$

$$\frac{\Sigma \vdash \tau_1 \leq \tau_2 \qquad \Sigma \vdash \tau_2 \leq \tau_3}{\Sigma \vdash \tau_1 \leq \tau_3}\ \textsc{Sub-Trans}$$

$$\frac{\begin{array}{c}\Sigma \vdash \beta_1 \sqsubseteq \beta_2 \\ \Sigma \vdash \gamma_1 <: \gamma_2 \qquad \Sigma \vdash \beta_1(\gamma_1)\ \mathbf{ok} \Sigma \vdash \beta_2(\gamma_2)\ \mathbf{ok}\end{array}}{\Sigma \vdash \beta_1(\gamma_1) \leq \beta_2(\gamma_2)}\ (\textsc{Sub-Name})$$

$$\frac{\Sigma \vdash \sigma_1 \leq \tau_1 \qquad \Sigma \vdash \tau_2 \leq \sigma_2}{\Sigma \vdash \tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2}\ (\textsc{Sub-Func})$$

$$\frac{\Sigma \vdash \tau \leq \sigma_1 \qquad \Sigma \vdash \tau \leq \sigma_2}{\Sigma \vdash \tau \leq \sigma_1 \wedge \sigma_2}\ (\textsc{Sub-}\wedge R)$$

$$\frac{\Sigma \vdash \tau_1 \leq \sigma}{\Sigma \vdash \tau_1 \wedge \tau_2 \leq \sigma}\ (\textsc{Sub-}\wedge L_1) \qquad \frac{\Sigma \vdash \tau_2 \leq \sigma}{\Sigma \vdash \tau_1 \wedge \tau_2 \leq \sigma}\ (\textsc{Sub-}\wedge L_2)$$

$$\frac{\Sigma \vdash \beta_1 \sqsubseteq \beta_2}{\Sigma \vdash \beta_1(\gamma_1) \wedge \beta_2(\gamma_2) \leq \beta_1(\gamma_1 \wedge \gamma_2)}\ (\textsc{Sub-Brand-}\wedge L_1)$$

$$\frac{\Sigma \vdash \beta_2 \sqsubseteq \beta_1}{\Sigma \vdash \beta_1(\gamma_1) \wedge \beta_2(\gamma_2) \leq \beta_2(\gamma_1 \wedge \gamma_2)}\ (\textsc{Sub-Brand-}\wedge L_2)$$

A.3 Evaluation

Definition A.3 (Matching case).

$$\begin{array}{l}\mathbf{select}_\Delta(\widehat{\theta}, \widehat{\beta}_i{}^{\,i\in 1..n}) \overset{\text{def}}{=} j \in 1..n, \\ \quad \text{where } \Delta \vdash \widehat{\theta} \sqsubseteq \widehat{\beta}_j \\ \qquad \text{and } \forall k \in 1..n\,.\, \Delta \vdash \widehat{\theta} \sqsubseteq \widehat{\beta}_k \Rightarrow \Delta \vdash \widehat{\beta}_j \sqsubseteq \widehat{\beta}_k\end{array}$$

Evaluation Relation

$\boxed{p \mid \Delta \longmapsto p' \mid \Delta'}$

$$\frac{}{\begin{array}{l} m\ \mathsf{brand}\ \beta_1(\gamma)\ \mathsf{extends}\ \beta_2\ \mathsf{in}\ p \mid \Delta \\ \quad \longmapsto\ p \mid \Delta, (\widehat{\beta}_1\ \mathsf{extends}\ \widehat{\beta}_2)\end{array}}\ (\textsc{E-Brand-Decl})$$

$$\frac{e \longmapsto_\Delta e'}{e \mid \Delta \longmapsto e' \mid \Delta}\ (\textsc{E-Expr})$$

$\boxed{e \longmapsto_\Delta e'}$

$$\frac{e_1 \longmapsto_\Delta e_1'}{e_1\, e_2 \longmapsto_\Delta e_1'\, e_2}\ (\textsc{E-App1}) \qquad \frac{e_2 \longmapsto_\Delta e_2'}{v_1\, e_2 \longmapsto_\Delta v_1\, e_2'}\ (\textsc{E-App2})$$

$$\frac{}{(\lambda x{:}\tau.\, e)\, v \longmapsto_\Delta [v/x]\, e}\ (\textsc{E-App-Abs})$$

$$\frac{e \longmapsto_\Delta e'}{e.\ell \longmapsto_\Delta e'.\ell}\ (\textsc{E-Proj1})$$

$$\frac{}{\widehat{\beta}(\overline{\ell_i = v_i}^{\,i\in 1..n}).\ell_k \longmapsto_\Delta v_k}\ (\textsc{E-Proj2})$$

$$\frac{e_i \longmapsto_\Delta e_i'}{\begin{array}{l}\widehat{\beta}(\ell_1 = v_1, \ldots, \ell_{i-1} = v_{i-1}, \ell_i = e_i, \ldots) \\ \quad \longmapsto_\Delta \widehat{\beta}(\ldots, \ell_i = e_i', \ldots)\end{array}}\ (\textsc{E-Brand-Cons})$$

$$\frac{e \longmapsto_\Delta e'}{\mathsf{case}\ e\ \mathsf{of}\ \{\cdots\} \longmapsto_\Delta \mathsf{case}\ e'\ \mathsf{of}\ \{\cdots\}}\ (\textsc{E-Case1})$$

$$\frac{\mathbf{select}_\Delta(\widehat{\theta}, \widehat{\beta}_i{}^{\,i\in 1..n}) = k}{\begin{array}{l}\mathsf{case}\ \widehat{\theta}(\overline{\ell = v})\ \mathsf{of}\ \{\overline{x_i\ \mathsf{as}\ \widehat{\beta}_i \Rightarrow e_i}^{\,i\in 1..n}\} \\ \quad \longmapsto_\Delta [\,\widehat{\theta}(\overline{\ell = v})\,/x_k\,]\, e_k\end{array}}\ (\textsc{E-Case2})$$