Brand Objects for Nominal Typing

Timothy Jones, Michael Homer, and James Noble

Victoria University of Wellington
New Zealand
{tim,mwh,kjx}@ecs.vuw.ac.nz

Abstract

Combinations of structural and nominal object typing in systems such as Scala, Whiteoak, and Unity have focused on extending existing nominal, class-based systems with structural subtyping. The typical rules of nominal typing do not lend themselves to such an extension, resulting in major modifications. Adding object branding to an existing structural system integrates nominal and structural typing without excessively complicating the type system. We have implemented *brand objects* to explicitly type objects, using existing features of the structurally typed language Grace, along with a static type checker which treats the brands as nominal types. We demonstrate that the brands are useful in an existing implementation of Grace, and provide a formal model of the extension to the language.

1998 ACM Subject Classification D.3.3 Classes and objects

Keywords and phrases brands, types, structural, nominal, Grace

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.198

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.1.1.4

1 Introduction

Most statically typed object-oriented languages use nominal subtyping. From Simula [5] and C++, through to Java, C♯ and Dart, an instance of one type can only be considered an instance of another type if the subtyping relationship is declared in advance, generally at the time the subtype is declared.

Some modern languages have adopted structural subtyping. In Go, for example, types are declared as interfaces, and an object conforms to a type if the object declares at least the methods required by the interface [41]. As well as Go's interfaces, Emerald is structurally typed [9], as is OCaml's object system [31] and Trellis/OWL [42]. Structural types have also been used to give types post-hoc to dynamically typed languages: Strongtalk originally supported structural types for Smalltalk [14], and Diamondback Ruby uses structural types for Ruby [23].

Given that nominal and structural typing both have advantages, there have been attempts to combine them both in a single language. The Whiteoak language [24] begins with Java's nominal type system and adds in support for structural types. Around the same time, Scala 2.6 [36] added structural types, again on top of a language with a nominal type system. The Unity language design similarly adds structural types onto a nominal class hierarchy [32].

The key argument of this paper is that adding nominal types to a structural type system requires a relatively smaller amount of effort. The corollary to this argument is that adding structural types into a nominal language – the direction of most existing approaches to the problem – does things backwards. This argument is based on our experience building a

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 198–221

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


branding mechanism in Grace, an object-oriented language with a standard structural type system [6, 7]. Both the static and dynamic nature of brands have been implemented using existing features of Grace, with minimal changes to the language.

We validate our argument by contributing:

- A practical design of *brand objects* for nominal typing on top of Grace's existing structural type system.
- An implementation of brand objects and a static nominal type checker in Hopper, a prototype implementation of Grace.
- Case studies of branding for various components of the language design in Hopper.
- A formal model of brands as nominal types as an extension to an existing model of a subset of the language.

The remainder of this paper proceeds as follows: Section 2 presents our motivation and design for branded types. Section 3 describes the implementation. Section 4 presents a series of case studies validating the utility of our branding system in the existing implementation. Section 5 formalizes our design, and proves soundness. Section 6 discusses alternative approaches, Section 7 discusses related work, and Section 8 concludes.

2 Brands

Nominal and structural typing both have advantages [32, 24]. Structural typing decouples an object's type – the set of methods to which it can respond – from the object's implementation (usually a class). Structural types can be declared at any time, in any part of the program, and still be relevant to any object with the appropriate interface. Any object that conforms to a structural type can be used wherever an instance of that structural type is required, even though the object's declaration did not declare that it implemented the type – among the reasons that Go adopted structural typing [41].

Being based solely on objects' interfaces rather than their implementations, structural types correspond to the conceptual model of object-oriented programming where individual objects communicate only via their interfaces, with their implementations encapsulated [19]. The clear separation between structural types and their implementing classes, and the ease of defining types independently from classes works well with gradual and pluggable typing [13], so programmers can begin by writing programs without types, and then add types later as the need increases.

On the other hand, nominal subtype relationships must be designed and declared by programmers, meaning they can capture programmers' intentions explicitly. Nominal subtyping can make finer distinctions between objects than structural subtyping: a structural system cannot distinguish between two different classes that have the same external interface, whereas a nominal system can distinguish between every implementation of every interface. Because nominal types can distinguish between different implementations (classes), compilers and virtual machines can optimize object allocation and method execution for particular implementations – for example, allocating machine integers and compiling arithmetic without any method dispatch.

As most languages are nominally typed, most of the major platforms for object-oriented languages (the Java Virtual Machine and the Common Language Runtime) are themselves nominally typed, so interoperability with VMs and languages is assisted by nominal typing. Pedagogically, nominal subtyping ensures every type has a name, so compilers and IDEs (especially their error messages) can refer to types by name, making teaching and debugging easier. Every nominal type has an explicit, unique, declaration in the program, a declaration that describes its relationships with all its supertypes, so class and type hierarchies can

be understood in a straightforward manner. These advantages are among the reasons that Strongtalk, for example, moved from structural to nominal typing [12, 14].

In order to implement brands, we have added *brand objects* to the Grace [6] programming language, extending its existing structural typing mechanism. A brand object represents a unique marker that can be applied to an object and subsequently detected, either statically or dynamically; the effects of these objects are similar to the branded types in Modula-3 [35]. We argue that, in comparison to adding structural typing to an existing class-based nominal system, adding nominal types to a structural type system requires only a very small change to the language, and can be achieved without any changes at all in a language with a relatively extensible type system. This is also demonstrated by the relatively simple additions of the formal model in Section 5, and by comparisons to other formalisms of branding systems.

Most nominal object-oriented type systems use a hierarchy of classes to define their types and name those types after the classes. Most branding systems use the same technique, whether or not they extend existing class-based languages. In such a language, brand types represents objects which have been created by their associated class, and conceptually include the interface of that class as well. In contrast, our brand objects have no associated class, and two objects branded with the same brand may have entirely distinct interfaces – we rely entirely on the existing structural type system to provide interface information. A brand type represents exactly those objects which have been branded by the underlying brand object, and no more.

We use three existing features of Grace in our implementation. Object annotations, where a newly constructed object is annotated with some other object, are used to explicitly brand objects (object annotations are a feature of Grace not yet discussed in the literature). A pattern [27] – which provides runtime pattern-matching facilities – acts as a brand's type, to test the presence of that brand on a given object. The dialect system [26] then allows the creation of a pluggable static type system which reasons about the patterns of brands bound to statically observable names as nominal types, and treats branded objects as inhabitants of these types.

2.1 Creating, Applying, and Using Brands

Consider a class hierarchy representing shapes, defining the concrete classes square and circle. In a strictly nominal system, an abstract class shape would form the root of this hierarchy, and create a common supertype for all concrete shape objects. In Grace, we might first define a Shape type:[1]

```
let Shape = type {
    at → Point
    area → Number
}
```

The Shape type describes the expected structure of a shape object. We could then define a class hierarchy which implements this interface, annotating the shape class as abstract with the keyword **is**:

```
class shape.at(location : Point) → Shape is abstract {
    method at → Point { location }
}
```

[1] Grace names returning types conventionally start in uppercase (Shape) while names returning objects (which may be fields, methods, or classes) start in lowercase (shape).

With concrete classes:

```
class square.at(location : Point) withLength(length : Number) → Shape {
    inherits shape.at(location)
    method area → Number { ... }
}

class circle.at(location : Point) withRadius(radius : Number) → Shape {
    inherits shape.at(location) ...
    method area → Number { ... }
}
```

Note that the classes are tagged with return types, as Grace classes are distinct from types. Moreover, all of these classes have the *same* return type, because their instances all have the same interface.

We could explicitly declare types for the objects created by the square and circle classes, by listing the signatures of the public methods in each class:

```
let Square = type { at → Point; area → Number }
let Circle = type { at → Point; area → Number }
```

These new types are identical to the Shape type defined above, and represent exactly the same set of objects. Structural types cannot distinguish between different objects with the same interface, either during static checking or at runtime.

In this paper, we introduce *brand objects* that can be used to make finer distinctions between objects – distinctions that correspond to standard nominal types. Brand objects are created by the brand method, which returns a new unique brand object. For example:

```
let aSquare = brand
```

will create a new brand object named aSquare. We can use this brand object to mark objects (e.g. those created by the square class) by annotating the class declaration with the brand:

```
class square.at(location : Point)
    withLength(length : Number) → Shape is aSquare { ... }
```

Brand objects have a Type method that returns a Grace pattern object that reifies the type of the brand. This lets us define distinct Square and Circle types by combining the structural Shape type with the types of the respective brands via Grace's type intersection operator (&).

```
let Square = Shape & aSquare.Type


let aCircle = brand
let Circle = Shape & aCircle.Type
```

Brands combined with structural interfaces produce Grace types that behave like nominal types. Square and Circle now define different types, rather than aliases of the same structural type.

This lets us go one step further: we can now declare that the square class returns an object of the Square type:

```
class square.at(location : Point)
    withLength(length : Number) → Square is aSquare { ... }
```

The instance's structural type is the same as before, but it carries the added information that it is branded as aSquare, making it an instance of the Square branded type as well. The brand aSquare is not a type, annotating the class with the brand is different from providing a return type, hence the appearance of both the aSquare brand and the Square type.

The combination of branded types with structural types follows the same type rules as other types, including subtyping. A branded object (with the appropriate brand) must be supplied where a branded object is expected:

```
def mySquare : Square = square.at(10 @ 50) withLength(20)
```

A branded object may be used anywhere an unbranded object with the same structure is expected:

```
def myShape : Shape = mySquare
```

But critically, an unbranded object cannot be supplied where a branded object is expected:

```
// Error: not an instance of Square
def myCircle : Square = circle.at(10 @ 50) withRadius(20)
```

If these declarations were repeated in another module (or even in the same module) then the repetition will create a different unique identifier and so represent a distinguishable, that is, different, brand and associated pattern object, regardless of the name the brand is bound to. The nominal aspect of the brand is its underlying object identity.

Brand objects, like types, can be reasoned about statically. For clarity on what we are treating statically in this paper we write "**let**" for all statically-known declarations, as a syntactic extension to the language. We discuss this change, which is not specific to branding, further in Section 3.

2.2 Extending Brands

Inheriting from an object that is branded causes the resulting object to have the same brand: this behavior is necessary for inheritance to preserve subtyping (required by the Grace specification [8]). Inheritance is the easiest mechanism for extending an existing brand, and provides a correspondence between class and (nominal) type, as in most nominally typed languages.

If we were to return to the shape class, and create a brand for it:

```
let aShape = brand
let Shape = aShape.Type & type { ... }
class shape.at(location : Point) → Shape is abstract, aShape { ... }
```

Now the whole shape hierarchy is branded, and the Shape type will only match objects created by the shape class, including those which inherit from it. The Square and Circle types remain subtypes of Shape and the square and circle classes inherit the aShape brand, just as if the classes were in a standard nominal typing hierarchy.

Brands need not conform to single-inheritance class hierarchies. Because brands are not inherently associated with an interface, and access to the brand object is all that is required to build an object which satisfies the brand type, any object can take advantage of multiple-subtyping without a multiple-inheritance mechanism by simply being branded with multiple brands. This is conceptually similar to a class implementing multiple interfaces in Java or C♯, providing a typing relationship without method reuse.

Brand objects also support the + operator, which creates a 'sub-brand' from two existing brands. Using this new brand is exactly the same as using the two parts together: branding an object with it causes the object to be branded with both the parts, and the brand's Type is the intersection of the patterns of both of the parts. Combining a brand with a new, anonymous brand, creates a unique sub-brand of the extended one. This behavior is included in the brand interface as the extend method.

2.3 Brands vs. Branded Types

The distinction between a brand (like aSquare) and a branded type (aSquare.Type, or Square) is crucial. Branded objects can only be created with access to the underlying brand. An untrusted object can safely be given access to the branded type, as this does not allow that object to fraudulently brand other objects. This can be achieved by exposing the branded type to other code as a public constant and keeping the brand object locally as a confidential field not accessible from the outside.

In Grace, a declared brand, like any other named declaration, is a method in an object; the name of a type (branded or not) is simply a request to the object declaring the type, and so Grace's existing visibility mechanism suffices to protect brands. In order to ensure no other class can be branded aSquare while allowing public access to the Square type, we would modify the brand declaration to be hidden:

```
let aSquare is confidential = brand
```

These declarations are public by default, so the type remains exported.

Access only flows in one direction: the brand object cannot be retrieved from the type, but access in the other direction is not limited, as the pattern object is available through the brand object with the Type method. The pattern object does not provide any privileged behavior, so it makes sense to provide uni-directional access between the objects rather than returning a pair from the brand constructor.

The three branding utilities – the brand method, the use of brands as annotations on object literals and classes, and the unique types they introduce – are the only additions Grace's structural type system requires in order to support nominal types. Moreover, using Grace's patterns and dialects, they are all achieved using existing functionality, with no brand-specific modifications to the language's syntax or semantics. In the next section, we discuss how this is achieved.

3 Implementation

We have implemented brands in Grace on top of Hopper, an existing prototype interpreter for the language. The core implementation is a single Grace module, extending the existing structural type checker described in [26]. We have also modified the language implementation to change **type** declarations to **let**, permitting any statically resolvable value to appear in the declaration.

All of the functionality specific to brands is implemented using existing features of Grace, provided in a *dialect* [26] with the necessary definitions. Every Grace module is written in a dialect that defines the methods that are in the local scope throughout that module. A dialect may also provide a check method, which is passed the AST of any module which uses it and may subsequently raise errors about the implementation of the module. The check method allows for a form of pluggable typing [13], which individual modules may opt in to. Modules written in the brand dialect will be checked, and the dynamic behavior of the

brands will work as expected in other modules, but the static checking is restricted to just those modules using the branding dialect.

Annotations are an existing feature of Grace, allowing objects to be attached to – and potentially transform – various language constructs, including singleton object constructors and classes. Hopper allows arbitrary expressions to appear in an annotation list, requiring only that the resulting object have an appropriate method for handling the construct that it annotates: for example, object annotations must have an annotateObject method. The method is implemented on brands so that it attaches the brand to the object's metadata, which is a weak set of objects associated with another object at runtime available through reflection. The metadata on a construct can be retrieved through a mirror object using the existing reflection interface. This means an object can be tested for a brand with:

```
mirrors.reflect(obj).metadata.has(aThing)
```

Grace's Pattern type, which provides an interface for testing objects against type-like objects as runtime [27], is used to build brand pattern objects. By inheriting from a standard abstract class that defines the basic implementation of patterns, the objects need only provide a concrete match method, which uses the reflection system mentioned above to inspect the metadata of the given object and discover whether the relevant brand is in the metadata set.

Internally, most of the brand object functionality is implemented in the preBrand class. This class is used to build the 'pre-brand' object aBrand. This pre-brand is local to the dialect, but the dialect makes the public type Brand for use outside of the dialect, which is aBrand's pattern object combined with the interface of brands.

```
let Brand = aBrand.Type & ObjectAnnotation & type {
    Type → Pattern
    extend → Brand
    +(other : Brand) → Brand
}
```

aBrand has the same implementation as other brands, but an object cannot appear in its own annotation list and so aBrand does not satisfy the Brand type. All other brands inherit from the same class that created aBrand, with the sole addition of being branded by aBrand, causing them to satisfy the Brand type.

```
method brand → Brand {
    object is aBrand { inherits preBrand.new }
}
```

These two definitions are included with the rest of the standard dialect definitions to provide a sensible set of default methods to any module which uses the dialect.

The dialect extends the existing structural type checker by including extra understanding of the Brand type and the result of requesting brand in its check method. Each creation of a brand object is considered distinct by the system, and this identity is tracked within the scope of its creation, as well as when it is exported by a **let** declaration. The existing structural rules are still enforced, including when structural types are paired with brand types. We formalize this combination of static typing in Section 5.

The replacement of **type** declarations with **let** is necessary to allow *any* dialect which is introducing a new type construct (rather than refining an existing one) to know what it should be reasoning about statically. This change is not specific to brands. The existing type declarations require that the value being declared be a statically-determinable structural type:

neither brands nor their pattern objects satisfy this definition, and neither will any other new value which a particular dialect treats as a static type. The definition of 'statically-known' is now determined on a module-by-module basis by the dialect a module is implemented in. The default and structural type checking dialects use the existing definition of static structural types, whereas the brand dialect extends this definition to include the new static brand values.

Matching against a brand's object identity provides no dynamic information about a brand's name, so that if a dynamic type error occurs involving a brand it cannot report the name of the brand that failed to match. Reporting type names is a standard problem in structural type systems [32] as types do not naturally have names. The use of static declarations in Grace also addresses this problem, by dynamically attaching name information to values – both brands and brand types, in our case – defined by a **let** declaration, which can be leveraged to generate better error messages. This behavior was already implemented for structural type declarations.

4 Case studies

Even in a structurally-typed language, some aspects will require more nominal semantics. This section presents applications of branding, mostly within the existing language implementation, replacing ad-hoc implementations with the branding mechanism.

4.1 Abstract Syntax Tree

An AST may contain many nodes with the same structure, but which must nevertheless be distinguished. This is a particularly important problem for Grace, as dialect check methods operate over the AST of the modules that they check. Nodes for a variable declaration and a constant definition will have a name, a value, and a type, but it is important that neither be mistaken for the other when they must be considered distinct. While the subtyping structure of an AST node is likely to be 'flat', brands allow overlaying distinguishing features on a range of otherwise-identical types.

We draw out two cases in particular from the AST of Grace source code, reflecting issues we have had ourselves in implementing the language. The **var** and **def** (variable and constant declaration) nodes mentioned in the previous paragraph have the same fundamental shape, while a **class** node has a superset of the methods of an **object** node, but is not a subtype of it. Before brands, AST nodes were 'stringly-typed', using a kind string field with the name of the node type, but this was an ad-hoc solution that sat outside of the type system. We can combine brands and types to avoid both of these issues: each kind of node now has both a structural interface and one or more nominal brands. Once we have created the relevant brands, the types can be constructed as:

```
// The common interface of both var and def nodes
let DeclNode = Node & type {
    name → String
    value → Expression
    pattern → Expression
}

let VarNode = aVarNode.Type & DeclNode
let DefNode = aDefNode.Type & DeclNode
```

The DeclNode type is purely structural, and before brands was the *only* type that applied to each of our nodes. VarNode, however, combines the structural type with the pattern of the aVarNode brand: to belong to the VarNode type, an object must have both the structural type and be branded as aVarNode.

```
class varNode.new(...) → VarNode is aVarNode { ... }

match(varNode.new(...))
    case { d : DefNode → print "A def!" }
    case { v : VarNode → print "A var!" }
```

Prior to brands, just as in our shapes example from earlier, the VarNode statement would 'fall into' the DefNode branch [11], because it is the first to appear and the structural type would match, and similarly a **def** node could be passed to a method expecting a **var** node without error. With brands, the DefNode branch does not match and the correct branch is given an opportunity to match, while both static and dynamic type checks will behave as desired.

4.2 Dialects

Dialects can be defined by expressing the checking as a series of rule blocks [26]. Rule blocks specify which nodes they apply to by typing their input, but this presents a problem to the type checker: if the input is stringly-typed, the type checker cannot determine what the type means and so cannot check the body of the rule. Misuses such as the spelling error below will not be caught until runtime, despite the rule being annotated with what appear to be types.

```
rule { vn : VarNode →
    if (vn.vallue.isEmpty) then {
        CheckerFailure.raise("All vars must be assigned to") forNode(vn)
    }
}
```

The extended reasoning of the branding allows the type checker to understand the combination of structural and nominal type.

A similar issue can occur when one type is a structural supertype of another. This situation arises in the case of class and object nodes, and the same resolution can be applied:

```
let ObjectNode = anObjectNode.Type & type {
    body → List<Node>
}

let ClassNode = aClassNode.Type & type {
    body → List<Node>
    name → String
}

class classNode.new(...) → ClassNode is aClassNode { ... }
```

Objects created by classNode will not be considered to belong to the ObjectNode type, notwithstanding that they possess all of the methods of object nodes. Before brands these nodes were distinguished by string fields found in all nodes, outside of the type system. Using

fields in this way is clearly suboptimal, particularly as it sits outside the protection of the type system.

4.3 Exceptions

Representations of runtime errors encode a degree of hierarchy, and must be both created and caught within this hierarchy. For example, a FileNotFoundError may be a specialization of IOError, which is itself a RuntimeError. An exception handler must be able to declare it wishes to trap all IOErrors, including specializations. In a nominal language such as Java this behavior maps naturally onto nominal class inheritance, with a handler for one exception type implicitly trapping all its subtypes by subsumption. In a structurally-typed language this relationship does not exist innately and must be created.

Grace's explicit exception hierarchy leverages the pattern-matching system for handlers. An *exception kind* is an object representing a kind of exception, and includes two methods. The refine method creates a new exception kind as a child of the receiver. The raise method creates an exception object, which is propagated up the stack until a handler is reached. All exception kind objects are patterns, matching any exception packet derived from itself or its refined descendants.

```
def FileNotFoundError = IOError.refine("File not found")
try {
    if (!exists(path)) then {
        FileNotFoundError.raise("{path} does not exist")
    }
} catch { e : IOError →
    print "An IO error occurred: {e}"
}
```

The catch block above will trap the exception raised in the try block because the exception kind FileNotFound was refined from IOError.

This system is reminiscent of brands and can be placed on firmer footing through their use. An ExceptionKind's match method delegates to the Type object of a brand, and its raise method creates an appropriately-branded exception packet. Omitting implementation details, the structure of the exception kind hierarchy can look like the following:

```
class exceptionKind.name(name : String) brand(aKind : Brand) → ExceptionKind {
    ...
    method refine(name : String) → ExceptionKind {
        exceptionKind.name(name) brand(aKind.extend)
    }
    method raise(message : String) → None {
        internal.raise(object is aKind { inherits exception })
    }
    method match(obj : Object) → MatchResult {
        aKind.Type.match(obj)
    }
}

let Exception = exceptionKind.name("Exception") brand(brand)
```

In this way brands provide a well-founded structure for an existing sui generis construct of the language. An exceptional behavior has been replaced with a consistent general-purpose approach that can be applied in user code elsewhere.

4.4 Singleton types

A singleton type is a type with only a single element, which may or may not be trivial. Singleton types are one way of adding nominal types into a structural language,[2] but we find it more advantageous to go in the other direction: to use brands as the means to add singleton types to a language without them. If a sentinel value unit is defined as an empty object, then its structural type is **type** {}, which is inhabited by every object. If Unit is to be a proper unit type, with unit as its only inhabitant, then we can define:

```
let theUnit is confidential = brand
let Unit = theUnit.Type

def unit is public = object is theUnit {}
```

As theUnit is not publicly available, other modules cannot brand other objects with it, and so unit will always be the only inhabitant proper of Unit.

Similarly, the empty type can also be constructed by taking the pattern of an anonymous brand, ensuring that no object can ever be branded by it and, by extension, ever be an instance of the resulting type.

```
let None = brand.Type
```

A brand need not be bound to a name to take its Type.

5 Formal Model

In this section, we model branded types by extending Tinygrace, an existing formal model of a subset of the Grace language [29]. A Tinygrace program is a set of type declarations and an expression to be evaluated. Unlike the gradual type system of the full Grace language, type information is always required: Tinygrace types are mandatory, and there is no Unknown (dynamic) type. Our extension makes two simple additions: branding objects with the **is** annotation marker, and creating brands with the brand declaration, as in the Grace implementation.

Brand declarations have no associated name or structural type information, but their associated types may be combined with structural information using the combinator &, and must be bound to a name or combined with other brands in order to be useful. In the following figures, changes to the existing model not related to structural typing are highlighted.

5.1 Syntax

The abstract syntax for the model is defined in Figure 1. The metavariable T ranges over static expression declarations; M over methods; O over object literals; C over case

[2] We discuss this approach in Section 6.

Syntax

$P ::= \overline{T}\ e$ *(Program)*

$M ::= \textbf{method}\ S\ \{\, e \,\}$ *(Method)*

$O ::= \textbf{object}\ \textbf{is}\ \overline{B}\ \{\, \overline{M} \,\}$ *(Object constructor)*

$C ::= \textbf{match}(e)\ \overline{\textbf{case}\ \{\, x : \tau \to e \,\}}$ *(Match-Case branch)*

$\tau ::= \textbf{type}\ \{\, \overline{S} \,\} \mid \mu X.\tau \mid X \mid (\tau \mid \tau) \mid (\tau\ \&\ \tau) \mid B.\mathsf{Type}$ *(Type)*

$S ::= m(\overline{x : \tau}) \to \tau$ *(Method signature)*

$e ::= x \mid e.m(\overline{e}) \mid O \mid C$ *(Expression)*

$B ::= \mathsf{brand} \mid B + B \mid X \mid \beta$ *(Brand expression)*

$E ::= \tau \mid B$ *(Static expression)*

$T ::= \textbf{let}\ X = E$ *(Static declaration)*

Contexts

$\Sigma ::= \cdot \mid \Sigma,\ X <: Y$ *(Subtyping context)*

$\Gamma ::= \cdot \mid \Gamma,\ x : \tau$ *(Typing context)*

Auxiliary Definitions

$\mathsf{or}(\tau) = \tau$
$\mathsf{or}(\tau,\ \overline{\tau}) = \tau \mid \mathsf{or}(\overline{\tau})$

$\mathsf{and}(\tau) = \tau$
$\mathsf{and}(\tau,\ \overline{\tau}) = \tau\ \&\ \mathsf{and}(\overline{\tau})$

Figure 1 Grammar for Tinygrace with branding extension.

expressions; x and y over variable names; X and Y over static expression alias names; τ over type expressions; S over method signatures, m over method names, and e over expressions.

We write $\overline{e}$ to indicate a possibly empty sequence of comma-separated expressions $e_1, \ldots, e_n$, as well as for method signature parameters $\overline{x : \tau}$ and type names $\overline{X}$, hiding the parentheses and **is** keywords when there are no values for them to delimit. We also write $\overline{S}$, $\overline{T}$, and $\overline{M}$ to indicate a possibly empty set of declarations $S_1 \ldots S_n$, $T_1 \ldots T_n$, and $M_1 \ldots M_n$ respectively, and $\overline{\textbf{case}\ \{\, x : \tau \to e \,\}}$ (or just $\overline{C}$) to indicate a non-empty sequence of case branches. As a Grace module is just an object, but the model does not allow static declarations in object bodies, a program is any pair of the form $\overline{T}\ e$. We follow Tinygrace in using Barendregt's variable convention [4, 44] that bound and free variables are distinct.

The set of type declarations $\overline{T}$ allows type aliasing as well as a mechanism for expressing recursive types without explicit folding. The declarations are resolved to explicitly recursive μ-types and substituted throughout the program to remove all aliases at runtime. μ-types are not a part of the concrete syntax, and can only arise from the normalization of type declarations.

$\boxed{T\ e\ \triangleright\ e}$

$$\frac{\overline{T} \vdash \overline{T\ \triangleright\ E'}}{\overline{T}\ e\ \triangleright\ [\overline{X \mapsto E'}]e}\ (\text{N-Prog})\quad \overline{T} = \overline{\textsf{let}\ X = E}, \text{ with } \overline{X} \text{ distinct}$$

$\boxed{\overline{T} \vdash \textsf{let}\ X = E\ \triangleright\ E'}$

$$\frac{\overline{T} \vdash \tau\ \checkmark}{\overline{T} \vdash \textsf{let}\ X = \tau\ \triangleright\ \mu X.\tau}\ (\text{N-Type-Decl})\quad \mu X.\tau \text{ contractive}$$

$$\frac{\overline{T} \vdash B\ \triangleright\ B'}{\overline{T} \vdash \textsf{let}\ X = B\ \triangleright\ B'}\ (\text{N-Brand-Decl})$$

$\boxed{\overline{T} \vdash B\ \triangleright\ B'}$

$$\frac{}{\overline{T} \vdash \textsf{brand}\ \triangleright\ \beta}\ (\text{N-Brand})\quad \beta \text{ fresh}$$

$$\frac{}{\overline{T} \vdash X\ \triangleright\ X}\ (\text{N-Name})\quad \textsf{let}\ X = B \in \overline{T}$$

$$\frac{\overline{T} \vdash B_1\ \triangleright\ B_1' \qquad \overline{T} \vdash B_2\ \triangleright\ B_2'}{\overline{T} \vdash B_1 + B_2\ \triangleright\ B_1' + B_2'}\ (\text{N-Plus})$$

Figure 2 Declaration normalization.

The major syntactic addition is the brand expression, with the metavariable B. The concrete syntax includes the brand constructor, sums of brands, and references to static brand declarations. Individual brands are resolved to a unique value in the set of names β. Like μ-types, β names are not part of the concrete syntax, arising only from the resolution of brand constructors. Ultimately, the names that brands are bound to become irrelevant, and they are identified solely by the uniqueness of their β name.

The remaining additions involve usage of brands: annotated objects and references to a brand's Type. The new metavariable E ranges over both brand expressions and types, allowing static declarations to refer to either.

5.2 Well-Formedness and Normalization of Declarations

The normalization judgments for programs, static declarations, and methods are given in Figure 2. A program $\overline{T}e$ is normalized into a single expression e' by $\overline{T}e \triangleright e'$ before it is analyzed or reduced. This normalization procedure corresponds to the well-formedness judgments of Tinygrace, making explicit how type declarations are transformed into anonymous recursive types inside the program expression.

Brand declarations B are normalized by $\overline{T} \vdash B\ \triangleright\ B'$, removing all occurrences of the brand constructor and replacing them with unique β identifiers to produce B'. Brand normalization also ensures well-formedness, as names in brand declarations must refer to an

$\boxed{\overline{T} \vdash \tau \checkmark}$

$$\text{(W-Struct)}\quad \frac{\overline{T} \vdash \overline{\tau_p} \checkmark \qquad \overline{T} \vdash \tau_r \checkmark}{\overline{T} \vdash \textbf{type}\,\{\,\overline{m(\overline{x : \tau_p}) \to \tau_r}\,\} \checkmark}\ \overline{m}\ \text{distinct}$$

$$\text{(W-Name)}\quad \frac{}{\overline{T} \vdash X \checkmark}\ \textbf{let}\ X = \tau \in \overline{T}$$

$$\text{(W-And)}\quad \frac{\overline{T} \vdash \tau_1 \checkmark \qquad \overline{T} \vdash \tau_2 \checkmark}{\overline{T} \vdash \tau_1 \,\&\, \tau_2 \checkmark}$$

$$\text{(W-Or)}\quad \frac{\overline{T} \vdash \tau_1 \checkmark \qquad \overline{T} \vdash \tau_2 \checkmark}{\overline{T} \vdash \tau_1 \mid \tau_2 \checkmark}$$

$$\text{(W-Rec)}\quad \frac{\overline{T}, \textbf{let}\ X = \tau \vdash \tau \checkmark}{\overline{T} \vdash \mu X.\tau \checkmark}$$

$$\text{(W-Brand)}\quad \frac{\overline{T} \vdash B \triangleright B'}{\overline{T} \vdash B.\mathsf{Type} \checkmark}$$

$\boxed{\Gamma \vdash M \checkmark}$

$$\text{(W-Meth)}\quad \frac{\Gamma, \overline{x : \tau_p} \vdash e_r : \tau_r}{\Gamma \vdash \textbf{method}\ m(\overline{x : \tau_p}) \to \tau_r\ \{\, e_r \,\} \checkmark}$$

Figure 3 Type and method well-formedness.

existing brand declarations in order to normalize. Note that normalization *only* transforms named declarations of brands, and other appearances of brand that appear in types or in the program expression are not resolved to some β. These 'dangling' brands can persist throughout the type-checking and execution of a program, but are considered distinct by subtyping and so cannot have any adverse effect on any of the remaining judgments.

Each type declaration must not resolve to itself (**let** $X = X$), and its interpretation as a tree must be *contractive*.

▶ **Definition 1.** A type tree is contractive if it corresponds to a finite series of μ-types or applications of & or |, so that all paths traversing any of those three operations terminates in a type literal.

For a type $\mu X.\tau$, this means that X may not appear in τ – either directly or through a reference to some other, mutually recursive declaration in $\overline{T}$ – except inside the body of a structural type literal. In the concrete syntax, this extends the set of invalid types to include declarations such as **let** $X = X \,\&\, Y$, or **let** $X = Y$; **let** $Y = X$.

Programs normalize to some expression e' when their set of declarations (both types and brands) normalize, and e' is the result of substituting the normalized declarations into the expression, with e' well-typed.

Type and method well-formedness are defined in Figure 3. Type well-formedness, $\overline{T} \vdash \tau \checkmark$, ensures that names inside τ only refer to existing type declarations. References to a brand's Type normalizes the brand, but discards the result, as only the well-formedness of the brand is required for the type to be well-formed. Dangling brands in Type references are retained, causing the whole type to be empty.

The well-formed judgment for methods $\Gamma \vdash M \checkmark$ just defers to the type judgment of the body of M in the scope of the method parameters.

5.3 Subtyping

The rules for type and signature subtyping are given in Figure 4, and are mostly standard. The subtyping relation is written $\Sigma \vdash \tau_1 <: \tau_2$, meaning a type τ_1 is a subtype of type τ_2 in the context of the assumption set Σ. The primary subtyping rule is Rule S-Struct, which states that two structural types $\textbf{type}\,\{\,\overline{S_1}\,\}$ and $\textbf{type}\,\{\,\overline{S_2}\,\}$ are in the subtyping relationship if, for a signature S_2, there is a signature S_1 with matching parameter types and return type in contravariant and covariant relationships respectively, and the subtraction of the matching signature from each type are also subtypes in the same direction. This forms an algorithmic approach to structural subtyping, removing a signature one at a time before terminating at Rule S-Top.

The assumption set models the otherwise coinductive nature of the recursive subtyping in a well-founded inductive setting. When comparing two recursive types with Rule S-Unfold the types may be unfolded but a subtyping relationship between the names bound by μ is added to the assumption set. If the same relationship is compared again, it succeeds through Rule S-Assum, modeling the infinite repetition permitted by coinduction. The assumption set need only contain the names of the types, as the well-formedness rules ensure that bound type names are unique to the type: if $\mu X.\tau$ appears in the program, any appearance of $\mu X.\tau'$ is guaranteed to have $\tau' = \tau$. Because the rules are interpreted inductively, the transitivity expressed in Rule S-Trans does not cause the judgment to degenerate.

The partial unfolding Rules S-Unfold-Left and S-Unfold-Right do not add to the assumption set, but the contractivity rules from the well-formedness rules ensure that the unfolded variable does not appear except inside of a type literal. This means that Rule S-Struct must apply in between, advancing the judgment.

The union and intersection types follow standard subtyping rules. The rules for subtyping of & are not complete with respect to the full language, as $\textbf{type}\,\{\,\overline{S_1}\,\}$ & $\textbf{type}\,\{\,\overline{S_2}\,\}$ is not equivalent to the union of $\overline{S_1}$ and $\overline{S_2}$ (an operation that requires combining signatures with the same name). This incompleteness does not affect the subset we are modeling.

The branding extension adds three rules to the subtyping judgment. Types of equivalent β names are subtypes, providing reflexivity for brands, and the patterns of sums of brands just defer to the intersection of the patterns of the summed brands. These rules only affect the relevant proofs by adding these extra cases into inductive reasoning.

The instance judgment $O \in \tau$, defined in Figure 5, means that a concrete object O is in the type τ. The judgment is defined directly in terms of subtyping on the concrete type of O, which the branded addition extends to include all of the brand patterns with the & combinator (the and auxiliary function is defined in Figure 1). We can interpret a type τ as a set $[\![\tau]\!]$ containing all of the concrete objects that are instances of that type, $\{O \mid O \in \tau\}$. We prove soundness of subtyping with respect to these set semantics.

First we require an inversion lemma on the instance judgment.

▶ **Lemma 2.** *If* $\textbf{object}\ \textbf{is}\ \overline{B}\ \{\,\overline{\textbf{method}\ S\ \{\,e\,\}}\,\} \in \tau$, *then* $\cdot \vdash \mathsf{and}(\textbf{type}\,\{\,\overline{S}\,\}, \overline{B.\mathsf{Type}}) <: \tau$

Proof. Trivial inversion of $O \in \tau$ by Rule S-In. ◀

We can now show soundness of subtyping, with respect to the set semantics of the types.

▶ **Theorem 3.** *If* $\cdot \vdash \tau_1 <: \tau_2$, *then* $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$.

$\boxed{\Sigma \vdash \tau_1 <: \tau_2}$

(S-Struct)
$$\frac{\Sigma \vdash \overline{\tau_{p2} <: \tau_{p1}} \qquad \Sigma \vdash \tau_{r1} <: \tau_{r2} \qquad \Sigma \vdash \textbf{type}\,\{\,\overline{S_1}\,\} <: \textbf{type}\,\{\,\overline{S_2}\,\}}{\Sigma \vdash \textbf{type}\,\{\,m(\overline{x_1 : \tau_{p1}}) \to \tau_{r1}\ \overline{S_1}\,\} <: \textbf{type}\,\{\,m(\overline{x_2 : \tau_{p2}}) \to \tau_{r2}\ \overline{S_2}\,\}}$$

(S-Top)
$$\frac{}{\Sigma \vdash \tau <: \textbf{type}\,\{\}}$$

(S-Assum)
$$\frac{}{\Sigma \vdash \mu X.\tau_1 <: \mu Y.\tau_2}\ X <: Y \in \Sigma$$

(S-Unfold)
$$\frac{\Sigma, X <: Y \vdash [X \mapsto \mu X.\tau_1]\tau_1 <: [Y \mapsto \mu Y.\tau_2]\tau_2}{\Sigma \vdash \mu X.\tau_1 <: \mu Y.\tau_2}\ X <: Y \notin \Sigma$$

(S-Unfold-Left)
$$\frac{\Sigma \vdash [X \mapsto \mu X.\tau]\tau <: \textbf{type}\,\{\,\overline{S}\,\}}{\Sigma \vdash \mu X.\tau <: \textbf{type}\,\{\,\overline{S}\,\}}$$

(S-Unfold-Right)
$$\frac{\Sigma \vdash \textbf{type}\,\{\,\overline{S}\,\} <: [X \mapsto \mu X.\tau]\tau}{\Sigma \vdash \textbf{type}\,\{\,\overline{S}\,\} <: \mu X.\tau}$$

(S-Or-Left)
$$\frac{\Sigma \vdash \tau <: \tau_1}{\Sigma \vdash \tau <: \tau_1 \mid \tau_2}$$

(S-Or-Right)
$$\frac{\Sigma \vdash \tau <: \tau_2}{\Sigma \vdash \tau <: \tau_1 \mid \tau_2}$$

(S-Or)
$$\frac{\Sigma \vdash \tau_1 <: \tau \qquad \Sigma \vdash \tau_2 <: \tau}{\Sigma \vdash \tau_1 \mid \tau_2 <: \tau}$$

(S-And)
$$\frac{\Sigma \vdash \tau <: \tau_1 \qquad \Sigma \vdash \tau <: \tau_2}{\Sigma \vdash \tau <: \tau_1 \,\&\, \tau_2}$$

(S-And-Left)
$$\frac{\Sigma \vdash \tau_1 <: \tau}{\Sigma \vdash \tau_1 \,\&\, \tau_2 <: \tau}$$

(S-And-Right)
$$\frac{\Sigma \vdash \tau_2 <: \tau}{\Sigma \vdash \tau_1 \,\&\, \tau_2 <: \tau}$$

(S-Trans)
$$\frac{\Sigma \vdash \tau_1 <: \tau_2 \qquad \Sigma \vdash \tau_2 <: \tau_3}{\Sigma \vdash \tau_1 <: \tau_3}$$

(S-Brand-Refl)
$$\frac{}{\Sigma \vdash \beta.\mathsf{Type} <: \beta.\mathsf{Type}}$$

(S-Brand-Left)
$$\frac{\Sigma \vdash B_1.\mathsf{Type} \,\&\, B_2.\mathsf{Type} <: \tau}{\Sigma \vdash (B_1 + B_2).\mathsf{Type} <: \tau}$$

(S-Brand-Right)
$$\frac{\Sigma \vdash \tau <: B_1.\mathsf{Type} \,\&\, B_2.\mathsf{Type}}{\Sigma \vdash \tau <: (B_1 + B_2).\mathsf{Type}}$$

Figure 4 Subtyping judgment.

Proof. Take any O such that $O \in [\![\tau_1]\!]$. For the exact type τ_o of O, $\cdot \vdash \tau_o <: \tau_1$ by Lemma 2. As $\cdot \vdash \tau_1 <: \tau_2$, $\cdot \vdash \tau_o <: \tau_2$ by the transitivity of subtyping, so $O \in \tau_2$ by Rule S-In. ◀

This property is not particularly interesting, as the instance rule is defined directly in terms of subtyping and so these outcomes are relatively obvious. We are more interested in the property of *method inclusion*, which ensures that when an expression is typed through the subtyping property, the resulting concrete object will have the necessary methods indicated by the type of the expression – this property is necessary for a structural subtyping system to be sound, and a proof for progress of well-typed terms under reduction relies on it. As not all types are structural, we consider any set of signatures in a structural supertype (equivalent to using a subsumption rule, which will be defined later).

$\boxed{O \in \tau}$

$$\text{(S-In)}\quad \frac{\cdot \vdash \mathsf{and}(\textbf{type}\,\{\,\overline{S}\,\}, \overline{B.\mathsf{Type}}) <: \tau}{\textbf{object}\ \textbf{is}\ \overline{B}\ \{\,\overline{\textbf{method}\ S\ \{\,e\,\}}\,\} \in \tau}$$

Figure 5 Instance judgment.

$\boxed{e \longmapsto e'}$

$$\text{(R-Recv)}\quad \frac{e \longmapsto e'}{e_s.m(\overline{e_p}) \longmapsto e'.m(\overline{e_p})} \qquad \text{(R-Prm)}\quad \frac{e \longmapsto e'}{O_s.m(\overline{O_p}, e, \overline{e_p}) \longmapsto O_s.m(\overline{O_p}, e', \overline{e_p})}$$

$$\text{(R-Req)}\quad \frac{\textbf{method}\ m(\overline{x : \tau_p}) \to \tau_r\ \{\,e_r\,\} \in \overline{M}}{\textbf{object}\ \textbf{is}\ \overline{B}\ \{\,\overline{M}\,\}.m(\overline{O_p}) \longmapsto [\mathsf{self} \mapsto \textbf{object}\ \textbf{is}\ \overline{B}\ \{\,\overline{M}\,\}, \overline{x \mapsto O_p}]e_r}\ |\overline{x}| = \left|\overline{O_p}\right|$$

$$\text{(R-Match)}\quad \frac{e \longmapsto e'}{\textbf{match}(e)\ \overline{\textbf{case}\,\{\,x : \tau \to e_c\,\}} \longmapsto \textbf{match}(e')\ \overline{\textbf{case}\,\{\,x : \tau \to e_c\,\}}}$$

$$\text{(R-Case)}\quad \frac{}{\textbf{match}(O)\ \textbf{case}\,\{\,x : \tau \to e_c\,\}\cdots \longmapsto [x \mapsto O]e_c}\ O \in \tau$$

$$\text{(R-Miss)}\quad \frac{}{\textbf{match}(O)\ \textbf{case}\,\{\,x : \tau \to e_c\,\}\ \overline{C} \longmapsto \textbf{match}(O)\ \overline{C}}\ O \notin \tau$$

Figure 6 Small-step semantics of reduction.

▶ **Theorem 4.** *If* $\cdot \vdash \tau <: \textbf{type}\,\{\,\overline{S_1}\,\}$*, then for any* $\textbf{object}\,\{\,\overline{\textbf{method}\ S_2\ \{\,e\,\}}\,\} \in [\![\tau]\!]$*, every* $S_1 \in \overline{S_1}$ *has a corresponding method* $S_2 \in \overline{S_2}$ *such that* $\cdot \vdash S_2 <: S_1$*.*

Proof. By induction over the derivation of $\cdot \vdash \tau <: \textbf{type}\,\{\,\overline{S}\,\}$. All of the rules which do not permit a structural type literal on the right of the subtyping judgment are irrelevant, and cannot become relevant in the derivation (because none of the relevant rules apply subtyping with a different type on the right side), so the derivation must ultimately terminate at Rule S-Top, by removing all of the signatures in the structural supertype after finding compatible signatures in the subtype through applications of Rule S-Struct. With case analysis on the last step, Rule S-Top is trivial, and the remaining relevant rules follow directly from the induction hypothesis. ◀

5.4 Semantics

The rules for the reduction relation $\longmapsto$ are given in Figure 6. The relation is written $e \longmapsto e'$, meaning that the expression e reduces to e' in a single reduction step. We write $\longmapsto^*$ for the

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

$$\text{(T-Var)}\quad \frac{}{\Gamma \vdash x : \tau}\ x : \tau \in \Gamma \qquad \text{(T-Sub)}\quad \frac{\Gamma \vdash e : \tau_1 \qquad \cdot \vdash \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}$$

$$\text{(T-Obj)}\quad \frac{\cdot \vdash \mathbf{type}\,\{\,\overline{S}\,\}\ \checkmark \qquad \Gamma, \mathsf{self} : \mathsf{and}(\mathbf{type}\,\{\,\overline{S}\,\}, \overline{B.\mathsf{Type}}) \vdash \overline{\mathbf{method}\ S\ \{\,e\,\}}\ \checkmark}{\Gamma \vdash \mathbf{object}\ \mathbf{is}\ \overline{B}\ \{\,\overline{\mathbf{method}\ S\ \{\,e\,\}}\,\} : \mathsf{and}(\mathbf{type}\,\{\,\overline{S}\,\}, \overline{B.\mathsf{Type}})}$$

$$\text{(T-Req)}\quad \frac{\Gamma \vdash e_s : \mathbf{type}\,\{\,\overline{S}\,\} \qquad m(\overline{x : \tau_p}) \to \tau_r \in \overline{S} \qquad \Gamma \vdash \overline{e_p : \tau_p}}{\Gamma \vdash e_s.m(\overline{e_p}) : \tau_r}\ |\overline{x}| = |\overline{e_p}|$$

$$\text{(T-Case)}\quad \frac{\Gamma \vdash e : \mathsf{or}(\overline{\tau_p}) \qquad \overline{\Gamma, x : \tau_p \vdash e_c : \tau_c}}{\Gamma \vdash \mathbf{match}(e)\ \overline{\mathbf{case}\,\{\,x : \tau_p \to e_c\,\}} : \mathsf{or}(\overline{\tau_c})}$$

Figure 7 Term typing judgment.

reflexive and transitive closure of $\longmapsto$. Object constructors are the only normal form of any expression. The judgment $e \longmapsto^* O$ represents a successful execution.

The rules for typing of terms is given in Figure 7. The typing judgment for expressions has the form $\Gamma \vdash e : \tau$, meaning that in the variable environment Γ, e has the type τ. The branding extension only modifies one rule that, like the instance judgment extension, folds the & combinator over a branded object's structural type and its declared brands' Types. We prove the soundness of the system through standard progress and preservation [22, 34], beginning with progress:

▶ **Theorem 5.** *If $\cdot \vdash e : \tau$, then $e \longmapsto e'$ or e is a term of the form O.*

Proof. By induction on the derivation of $\cdot \vdash e : \tau$, with a case analysis on the last step. Rule T-Var is irrelevant, as $\Gamma = \cdot$. Rule T-Sub and Rules T-Req and T-Case for congruence follow from the induction hypothesis. Rule T-Req for computation ensures that an appropriate method with appropriate parameter cardinality exists (guaranteed through subsumption with Theorem 4), and Rule T-Case for computation ensures that either there is more than one case or, when there is one case, the object is guaranteed to match that final case. The added branding rule in Rule T-Obj is as trivial as the existing object rule, as a branded object is still a value and so is still in normal form. ◀

Next we require substitution preservation.

▶ **Lemma 6.** *If $\Gamma, x : \tau' \vdash e : \tau$ and $\Gamma \vdash O : \tau'$, then $\Gamma \vdash [x \mapsto O]e : \tau$.*

Proof. By induction on the derivation of $\Gamma, x : \tau' \vdash e : \tau$. The cases are straightforward examinations of substitution. Branding only affects objects, which may have brand annotations, but this is not affected by substitution. ◀

The following lemma takes the observation that, for the subtyping relationship $\cdot \vdash \tau_1 <: \mathsf{or}(\overline{\tau_2})$, at least one of the types in $\overline{\tau_2}$ must also be a supertype of τ_1, and places it in the context of preservation for Rule R-Miss, where subtracting a type from the series of unions preserves the union as a supertype of τ_1 if the subtracted type was not a supertype of τ_1.

▶ **Lemma 7.** *If* $\cdot \vdash \tau_1 <: \tau_2 \mid \tau_3$ *and* $\cdot \nvdash \tau_1 <: \tau_2$*, then* $\cdot \vdash \tau_1 <: \tau_3$*.*

Proof. By induction on the derivation of $\cdot \vdash \tau_1 <: \tau_2 \mid \tau_3$. Rule S-Or-Left cannot apply, and the remaining rules must ultimately delegate to Rule S-Or-Right to be well-founded. Branding just adds a case for Rule S-Brand-Left. ◀

Now we have the tools for a proof of preservation:

▶ **Theorem 8.** *If* $\cdot \vdash e : \tau$ *and* $e \longmapsto e'$*, then* $\cdot \vdash e' : \tau'$ *where* $\cdot \vdash \tau' <: \tau$*.*

Proof. By induction on the derivation of $e \longmapsto e'$. The congruence rules follow straightforward induction, and the computation rules are derived from the two previous lemmas, using Lemma 6 for Rules R-Req and R-Case, and Lemma 7 for Rule R-Miss. Branding has a minimal impact on both reduction and typing of terms, and so does not pose a problem here. ◀

And finally we have type soundness.

▶ **Theorem 9.** *If the expression* e *is well-typed with* $\cdot \vdash e : \tau$*, and the reduction* $e \longmapsto^* e'$ *results in* e' *a normal form, then* e' *is in the form* O *where* $\cdot \vdash O : \tau'$ *with* $\cdot \vdash \tau' <: \tau$*.*

Proof. Type soundness follows immediately from the two previous theorems. ◀

6 Discussion

In order to demonstrate the *relative* simplicity of our branding additions, we compare our formal model to formalizations of the other branding systems Unity [32], and the Tagging Language [25]. Objective measures of the complexity of type systems are difficult, but we can produce a simple comparison on the number of formal rules that are not significant to branding, alongside the rules that are. The outcome is the table in Figure 8. We omit Whiteoak [24] because it does not provide a formal model, but the Whiteoak design has almost as many additions to the syntax as Tinygrace makes overall. Both Tinygrace and the branded extension have a relatively large well-formedness overhead, but they have the same number of subtyping rules as Unity, and brands make a substantially smaller impact on typing and reduction in Tinygrace than in either Unity or even the significantly smaller Tagging Language (which lacks objects, fields, classes, and dynamic dispatch). The larger number of well-formedness rules in Tinygrace seems to stem from making less assumptions in the translation from type (and brand) declarations to recursive type (and brand) expressions.

Another option to support nominal types in any structural system by the addition of unique method names into the type, and empty implementations of these methods into the objects which are expected to fulfill this type. While the 'phantom method' approach works in theory, it is difficult to implement in a way that preserves the necessary encapsulation goals of nominal typing. If the methods are provided manually, the developer must provide method names that will not be used anywhere else in the program, and the encapsulation is trivially bypassed by adding the appropriate methods to other, external objects. If the methods are produced automatically, then either the branding mechanism cannot be private, or the automatic names must be indexed by something (presumably the module in which

	Tinygrace	Unity	Tagging
Syntax	7 + 4	9 + 5	5 + 5
Well-formedness	8 + 5	4 + 2	3 + 2
Subtyping	13 + 3	13 + 3	2 + 2
Term typing	5 + 1	9 + 2	6 + 4
Reduction	7 + 0	14 + 4	3 + 4
Total	40 + 13	49 + 16	19 + 17

Figure 8 Comparison of rule modifications required by branding.

the branding appears), in which case brands cannot be shared because no other module may perform the same branding. A unique singleton or empty type as the return type of a common 'branding' method in a type is another approach, but it suffers from the same problem in that the brand and type cannot be exposed separately. Neither mechanism can simultaneously service both public construction and private implementation. Our brands, in contrast, provide a fine-grained mechanism for providing access to the branding mechanism components.

Branding provides a partial solution to Boyland's gradual guarantee in gradually typed code [11], as testing an object against a brand pattern at runtime is always definitive, and is not affected by type annotations. As an extension, branding is not pervasive among objects, and so using brand patterns is only applicable to objects which have been explicitly branded. Removing reified types from the language (another proposed solution) while retaining the existing object instance rules would remove the consistency of brand patterns (as both static entities and runtime objects) alongside structural types.

Compared to standard nominal class declarations, the branding mechanism is necessarily verbose, requiring a manual separation of the brand from its type (mirroring the separation between classes and structural types in Grace). This verbosity is mostly a product of the fact that brands have been implemented without modifying the language syntax or semantics, but it also serves a purpose in demonstrating that it is not the natural mechanism for typing in Grace: structural typing is sufficient for most purposes, and it is only special cases (as seen in Section 4) where the manual separation of brand and pattern that branding should apply. It is conceivable that a more terse mechanism for direct class/type declarations could exist:

```
class Shape.new is nominal { ... }
```

Adding nominal classes directly defies Grace's design goal of maintaining a separation of type and implementation, however [6].

7 Related Work

The dichotomy between structural and nominal subtyping has been studied from the earliest applications of types to object-oriented languages [10]. Simula, the first object-oriented language, is nominally typed: a subclass must be explicitly declared as inheriting (being prefixed) by its superclass [5]. Most object-oriented languages (C++, Java, C♯, etc) followed Simula's lead, although OCaml [31] supports structural subtyping for objects, as does Go [41].

Most early theoretical analysis of type systems for object-oriented languages used structural types [20, 17, 15, 40]. Later references such as Palsberg and Schwartzbach [38], Bruce [16], and Pierce [39] discuss structural and nominal (sub)typing, but they do not address the question of how both kinds of types can best be integrated into a single, practical, language design.

Our notion of nominal brands on structural types originated in Modula-3 [35]. Record types in Modula-3 generally use structural equivalence, but can be annotated with a brand to give nominal equivalence. Modula-3 brands can also be given explicitly, e.g. for type safety between programs or across networks. Even with structural equivalence, Modula-3 record types do not support subtyping: there is no type relationship between a record type with a particular set of fields, and a second record type with a subset (or superset) of those fields – only between two record types whose field types are identical. Modula-3's object types are "essentially SIMULA classes" [18] and, like SIMULA, use nominal subtyping. Neither Cardelli et al.'s formalization of the Modula-3 type rules [18], nor Abadi's Baby Modula-3, [1], nor the *Theory of Objects* [2] model Modula-3's branded types: rather, the formal model we present here is the first we know of that shows how Modula-3–inspired brands can allow a language to support both structural and nominal subtyping.

Malayeri and Aldrich's Unity language [32, 33] is a more recent clean-sheet language design that aims to support both nominal and structural typing. Unity also uses brands to support nominal typing, but brands in Unity are essentially nominal classes – unlike Modula-3 or Grace brands, which are annotations on structural types and objects respectively. Unity's brands define the core object hierarchy in a Unity program, potentially extending a superbrand and defining fields and methods in the exactly same way that in, say, Java classes potentially extend a superclass and define the fields and methods of their instances. Unity objects are created by instantiating a brand, again just as Java (or SIMULA) objects are created by instantiating a class. Brands give Unity a nominal core, to which structural types are then added, in contrast to the approach presented here, which adds nominal brands on top of structural types. Unity draws on structural types to support external multimethods and fields defined outside objects. While Grace does not not support multimethods, similar effects can often by obtained by pattern matching, which Grace supports using both structural and nominal types. Unity was modeled formally, but not implemented.

Glew's Tagging Language [25] introduces 'tags', which are conceptually very similar to our brand objects, in the context of type dispatch. Tags can be used to implement class- and exception-casing in much the same way as brands. The underlying type system is not structural, and is populated by primitive sequence and function types instead. The language formalism goes into depth on the existence of tags at runtime, including populating the heap and runtime matching.

Gil and Maman's Whiteoak [24] in many ways takes a more pragmatic approach than Unity to combining structural and nominal types. Where Unity is a clean-sheet design, Whiteoak adds structural types to Java. Whiteoak uses the '`struct`' keyword (reserved in Java) to define structural types, in practice very similar to Java interfaces except that struct's subtyping is, of course, structural. Whiteoak also has some features for post-hoc object extension, and a form of trait composition. Unlike Unity, Whiteoak has been implemented, and a type checking algorithm is described, although the type system has not been formalized.

Beginning in Scala 2.6, Scala supports structural types as refinements of the top type `AnyRef` [36]. Structural types may appear wherever Scala types are expected, and generally may take part in Scala's rich and multifaceted type system. The formalizations of Scala's type systems, νObj [37], FS_{alg} [21], and μDOT [3] are nominal, and do not incorporate

structural types (Scala's "refinements"). Philosophically, Scala, Whiteoak, and Unity share a single approach: adding structural types to an existing nominal system. Our approach is the opposite.

Brands and structural typing have also been used in more practical languages. Trellis/OWL was based on structural types [42] although without brands. Strongtalk [14] is an optional (AKA pluggable) type system for Smalltalk: in the original version of Strongtalk, the types were structural with optional brands, again very similar to our design, although a later version of Stongtalk abandoned brands and adopted declared subtyping and matching relationships [12]. Diamondback Ruby adopted a type system very similar to Strongtalk's to type check Ruby programs [23]. Diamondback Ruby does not use brands, although it employs both nominal types generated from classes, and structural types to describe individual objects. Many other efforts to add types to dynamic languages, such as Typed Scheme/Racket, provide a set of type combinators such as intersection or union which allow the build up of more complicated type aliases [43]. Typed Racket in particular takes advantage of the language's underlying extensibility to include the type system as a library, rather than in the language.

Trademarks have been proposed for ECMAScript 6 [28], which provide a very similar model of branding for the language. Trademarks are also split between a branding object and a 'guard', the latter of which is conceptually a type, with the same mechanism of hiding the branding object while exposing the guard to prevent fraudulent branding. As a dynamically-typed language, combining static reasoning about trademarks with a static structural type system would be useful.

The implementation of brand objects, with the brand type accessible through a method on the brand itself, is reminiscent of the path dependent types in μDOT [3]. Following a chain of statically-known values is required in order to resolve the type statically, and is part of the requirement for the **let** construct. μDOT focuses more on strictly defined associated types, whereas Grace currently does not allow type literals to include other type declarations inside of themselves.

Recent work includes the Tagged Objects of Lee et al. in Wyvern, which adds nominal tags on top of an existing structural type system [30]. This approach focuses on the type theory of tags, and provides new primitive type and matching constructs as an extension to the language, with new static typing rules. This differs from our branding, which introduces the nominal types through existing language features including dialects and runtime reflection.

8 Conclusion

In this paper, we have described how we added nominal types (via brand objects) as a minimal extension to Grace's structural type system. We have demonstrated this extension on top of the existing implementation Hopper, drawing on Grace's pattern matching and dialects. We have provided several case studies of brands in the existing implementation, and have modeled the new type system and proved it sound. The key advantage of our approach is that brands are a much smaller addition to a preexisting structural type system than structural types are to a preexisting nominal type system, and required only a minimal change to the underlying language. So, if you are going to design a language that combines nominal and structural typing, our strong advice is to follow Modula-3: start with a structural system and then add nominal types, rather than to follow Unity, Whiteoak and Scala, which start with a nominal system and then add what amounts to another entire (structural) type system alongside.

References

1 Martin Abadi. Baby Modula-3 and a theory of objects. *Journal of Functional Programming*, 4(2):249–283, 1994.
2 Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.
3 Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In *OOPSLA*, 2014.
4 Henk Barendregt. *The Lambda Calculus*. North-Holland, revised edition, 1984.
5 G. M. Birtwistle, O. J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Studentlitteratur, 1979.
6 Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. Grace: the absence of (inessential) difficulty. In *Onward!*, pages 85–98, 2012.
7 Andrew P. Black, Kim B. Bruce, Michael Homer, James Noble, Amy Ruskin, and Richard Yannow. Seeking Grace: a new object-oriented language for novices. In *SIGCSE*, pages 129–134, 2013.
8 Andrew P. Black, Kim B. Bruce, and James Noble. The Grace programming language (draft specification version 0.3.1303). `http://gracelang.org/documents/grace-spec031303.pdf`, 2013.
9 Andrew P. Black, Eric Jul, Norman Hutchinson, and Henry M. Levy. The development of the Emerald programming language. In *History of Programming Languages III*. ACM Press, 2007.
10 Andrew P. Black and Jens Palsberg. Foundations of object-oriented languages – workshop report. *SIGPLAN Notices*, 29(3):3–11, 1994.
11 John Tang Boyland. The problem of structural type tests in a gradual-typed language. In *FOOL*, New York, NY, USA, 2014. ACM.
12 Gilad Bracha. The Strongtalk type system for Smalltalk. In *OOPSLA Workshop on Extending the Smalltalk Language*, 1996.
13 Gilad Bracha. Pluggable Type Systems. OOPSLA workshop on revival of dynamic languages, 2004.
14 Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA*, 1993.
15 Kim B. Bruce. A paradigmatic object-oriented programming language: design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.
16 Kim B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.
17 L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.
18 Luca Cardelli, James E. Donahue, Mick J. Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 type system. In *POPL*, pages 202–212, 1989.
19 William R. Cook. On understanding data abstraction, revisited. In *OOPSLA*, 2009.
20 William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 125–135, 1990.
21 Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for Scala type checking. In *MFCS*, pages 1–23, 2006.
22 Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.
23 M. Furr, J.-H. An, J. Foster, and M.J. Hicks. Static type inference for Ruby. In *SAC*, pages 1859–1866, 2009.
24 Joseph Gil and Itay Maman. Whiteoak: Introducing structural typing into Java. In *OOPSLA*, 2008.
25 Neal Glew. Type dispatch for named hierarchical types. In *ICFP*, New York, NY, USA, 1999. ACM.

26 Michael Homer, Timothy Jones, James Noble, Kim B. Bruce, and Andrew P. Black. Graceful dialects. In *ECOOP*, pages 131–156, 2014.

27 Michael Homer, James Noble, Kim B. Bruce, Andrew P. Black, and David J. Pearce. Patterns as objects in Grace. In *DLS*, New York, NY, USA, 2012. ACM.

28 Waldemar Horwat and Mark Miller. ES6 Strawman: Trademarks. `http://wiki.ecmascript.org/doku.php?id=strawman:trademarks`, 2011.

29 Timothy Jones and James Noble. Tinygrace: A simple, safe and structurally typed language. In *FTFJP*. ACM, New York, NY, USA, 2014.

30 Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. A theory of tagged objects. In *ECOOP*, 2015.

31 Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 3.12 documentation and user's manual, 2011.

32 Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *ECOOP*, 2008.

33 Donna Malayeri and Jonathan Aldrich. Is structural subtyping useful? An empirical study. In *ESOP*, 2009.

34 John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *TOPLAS*, 10(3), 1988.

35 Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

36 Martin Odersky. *The Scala Language Specification: Version 2.9*. Programming Methods Laboratory, EPFL, Switzerland, 2011.

37 Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *ECOOP*, pages 201–224, 2003.

38 Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, Chichester, 1994.

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

40 Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4:207–247, 1994.

41 Rob Pike and The Go Team. The Go programming language specification. `http://golang.org/ref/spec`, 2014.

42 Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/OWL. In *OOPSLA*, 1986.

43 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *POPL*, 2008.

44 Christian Urban, Stefan Berghofer, and Michael Norrish. Barendregt's variable convention in rule inductions. In *CADE*, pages 35–50, 2007.