# Brand Objects for Nominal Typing

Timothy Jones, Michael Homer, and James Noble

Victoria University of Wellington

{tim,mwh,kjx}@ecs.vuw.ac.nz

July 8, 2015

# This Talk

More tagged types

- The intersection of first-class structural and nominal types
- Language design issues

Grace

- Structurally typed
- Classes are only sugar

*Brand Objects*

- First-class nominal types
- Both dynamic and static behaviour
- Access managed with standard OO encapsulation

# Motivation

*"structural types correspond to the conceptual model of object-oriented programming where individual objects communicate only via their interfaces, with their implementations encapsulated"*

# Motivation

*"structural types correspond to the conceptual model of object-oriented programming where individual objects communicate only via their interfaces, with their implementations encapsulated"*

— Jones et al.

# Structural Typing

Only interface matters

```
let Person = type {
    name → String
}

def me : Person = object {
    method name → String { "Tim" }
}
```

# Structural Typing

Types are implicit

```
let Person = type {
    name → String
}

def me = object {
    method name → String { "Tim" }
}
```

# Motivation

*"often frameworks require inheriting from a specific class with specific hidden state"*

# Motivation

*"often frameworks require inheriting from a specific class with specific hidden state"*

— Sam Tobin-Hochstadt

# Why Grace

Why not address this problem using Racket?

- First-class classes
- Type erasure

# Why Grace

Why not address this problem using Racket?

- First-class classes
- Type erasure
- Dialects aren't `#lang`

# Motivation

*"I do not see how a number object in Grace can for sure recognize another number object in the first place"*

— Marco Servetto

# Brands as Hybrids

Class names equipped with extra structural information

```
class Window { ··· }

method scrollUp(win : Window { scrollBar → ScrollBar }) {
  win.scrollBar.position := 0
}
```

No structural type without a class name

- Top type is Object {}

# Brand Objects

Objects are not associated with a class

```
let ScrollWindow = Window & type { scrollBar → ScrollBar }

method scrollUp(win : ScrollWindow) {
  win.scrollBar.position := 0
}
```

Structural types are a separate construct

- Top type is **type** {}

# Reification

Types are reified as objects at runtime

**instanceof** checks performed with a match() method

```
if(Person.match(me)) then {
    print "I'm a person!"
}
```

Type-safe branching with match() case()

```
match(animal)
    case { dog : Dog → · · · }
    case { cat : Cat → · · · }
```

# Reification

Types are reified as objects at runtime

# ~~Reification~~

Types are objects

# ~~Reification~~

Types are objects

- We just happen to (occasionally) reason about them statically

# Brand Objects

We can build new kinds of objects and treat them as types too

- Brands are just objects: no language extensions needed

# Constructing a Brand

The brand method

```
let aSquare = brand
```

# Applying Brands

Branding an object

```
object is aSquare {
    inherits shape.at(2 @ 5)
    method area → Number { ··· }
}
```

Uses the existing annotation system

# Applying Brands

Branding a class

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

## Brand Types

Brand objects are distinct from their corresponding types

```
let Square = aSquare.Type

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

# Brand Types

Brand objects are distinct from their corresponding types

```
let Square = aSquare.Type & Shape

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

Combined with structural types to build ‘full’ nominal types

# Inheritance

Inheritance preserves subtyping

```
def mySquare : Square = object {
    inherits square.at(2 @ 5) withLength(20)
}
```

# Extending Brands

Branding the whole shape hierarchy

```
let aShape = brand
let Shape = aShape.Type

let aSquare = aShape.extend
let aCircle = aShape.extend

def mySquare : Square = object is aSquare {}
```

## Extending Brands

Branding the whole shape hierarchy

```
let aShape = brand
let Shape = aShape.Type

let aSquare = aShape.extend
let aCircle = aShape.extend

def mySquare : Shape = object is aSquare {}
```

# Extending Brands

Multiple subtyping

```
let aSquaredCircle = aSquare + aCircle

def mySquare : Square = object is aSquaredCircle { ··· }
def myCircle : Circle = mySquare
```

# Extending Brands

Works in both directions

```
let SquaredCircle = aSquaredCircle.Type

def both : SquaredCircle = object is aSquare, aCircle {}
```

# Permissions

See the ECMAScript strawman for Trademarks™

> *"Given the brander one can readily create a guard. On the other hand, one cannot obtain the brander given just the guard of a trademark. Thus the brander of a trademark is a capability."*

# Permissions

Standard object encapsulation provides necessary restrictions

```
let aSquare is confidential = brand

let Square is public = aSquare.Type
```

Modules are just objects

# Branding Dialect

Is our language extensibility powerful enough to introduce radically new type constructs?

# Branding Dialect

`brand` method, with dynamic behaviour

Accompanying static checker

All branding features also provided by the language

- Dialect checking
- Annotations
- Encapsulation
- First-class type interface

# Types as a Library

Building types using existing language constructs

- Interesting for existing dynamically-typed languages

# Types as a Library

We claim it would be significantly more difficult to add structural types to an existing nominally-typed (class-based) system

- Syntax
- Infrastructure
- Reflection

More than just the sum

# 'Nominal' Typing

Names remain irrelevant

- Only the identity of the brand matters

Must be bound to a name to be useful

- Static checker tracks brand identities as locally-bound definitions

Names are useful!

- This is true for structural types as well
- Mitigated with a little **let** magic

# ‘Nominal’ Typing

There is exactly one use case for an anonymous brand

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

# Static Reasoning

Dialect reasons about brands it can statically resolve

Observes each request to brand and introduces a new type

```
let aSquare = brand
```

The brand method returns a value of type Brand

# Static Reasoning

Behind the scenes, the type of each application is different

$\textbf{let}\ \mathsf{aThing}_1 : \mathsf{Brand} = \mathsf{brand}$

$\textbf{let}\ \mathsf{aThing}_2 : \mathsf{Brand} = \mathsf{brand}$

# Static Reasoning

Behind the scenes, the type of each application is different

$$\mathbf{let}\ \mathsf{aThing}_1 : \mathsf{Brand}\langle\mathsf{aThing}_1\rangle = \mathsf{brand}$$

$$\mathbf{let}\ \mathsf{aThing}_2 : \mathsf{Brand}\langle\mathsf{aThing}_2\rangle = \mathsf{brand}$$

This isn't really expressible in the syntax

- (But it doesn't need to be)

# Static Reasoning

Brand is a regular type, and its values can be reasoned about

```
method using(aThing : Brand) {
    let Thing = aThing.Type

    def thing : Thing = object is aThing { ··· }

    ...
}
```

## Static Reasoning

We don't have dependent types

```
method make(aThing : Brand) → aThing.Type {
    object is aThing { ··· }
}
```

Lee et al.

# Formalisation

Extension to Tinygrace

# Normalization

$$\frac{\overline{T} \vdash \tau \checkmark}{\overline{T} \vdash \textbf{let } X = \tau \;\triangleright\; \mu X.\tau} \quad \mu X.\tau \text{ contractive}$$

$$\frac{\overline{T} \vdash B \;\triangleright\; B'}{\overline{T} \vdash \textbf{let } X = B \;\triangleright\; B'}$$

$$\frac{}{\overline{T} \vdash \mathsf{brand} \;\triangleright\; \beta} \quad \beta \text{ fresh}$$

$$\frac{}{\overline{T} \vdash X \;\triangleright\; X} \quad \textbf{let } X = B \in \overline{T}$$

$$\frac{\overline{T} \vdash B_1 \;\triangleright\; B_1' \qquad \overline{T} \vdash B_2 \;\triangleright\; B_2'}{\overline{T} \vdash B_1 + B_2 \;\triangleright\; B_1' + B_2'}$$

# Modifications

Existing + Branding

| | 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 |

# Soundness

Branding has a minimal impact on soundness

# Language Design Questions

What is a ‘type’?

The **let** definition

- Use cases feed back into language design

# Class-name types

Encode the one-brand-per-class pattern as an annotation

**class** Shape.new **is** nominal $\{ \cdots \}$

# Conclusion

Types are whatever you want them to be!

- So long as you can work out static reasoning for them

Libraries of types

- With extensible language features

Easier to start with a structurally-typed base

- Classes aren't necessary for nominal typing

# Links

tim@ecs.vuw.ac.nz

http://drops.dagstuhl.de/opus/volltexte/2015/5231/

http://ecs.vuw.ac.nz/~tim/publications/talks/ecoop2015.pdf

Kim's talk tomorrow

# Extra Slides

# The AST

Type hierarchy does not match node hierarchy

Custom pattern objects are not types

```
rule { vn : Var →
    !vn.vallue.isImplicit
}
```

# Exceptions

All exception objects have the same interface

We want to have a standard catch construct

```
catch { e : IOError →
    print "An IO error occurred: {e}"
}
```

Moves an internal implementation into the language

# Singleton and Empty Types

The empty structural type is the top type

We can build a proper unit type by branding exactly one object

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

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

The Type of an anonymous brand is guaranteed to be empty

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

# Brands as a dialect

`brand` constructor, with dynamic behaviour

Accompanying static checker

Remainder of features provided by the language

- Dialect checking
- Annotations
- First-class type interface

# Brands as a case study

Is our language extensibility powerful enough?

**let** is new

- Brands aren't types
- Unclear semantics for **type** declarations

# Pre-Branding

All brands are themselves branded

There must be some initial 'pre-brand'

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

class preBrand.new → BrandInterface { · · · }
```

# Pre-Branding

The brand constructor puts it all together

```
let aBrand = preBrand.new
let Brand = aBrand.Type & BrandInterface

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

# Matching

Each brand is equipped with a weak set

When an object is branded, it is placed in the set

When asked to match() against an object, a brand's Type checks its presence in the set

# 'Nominal' Typing

Names remain irrelevant

- Only the identity of the brand matters

Must be bound to a name to be useful

- Static checker tracks brand identities as locally-bound definitions

Names are useful!

- This is true for structural types as well
- Mitigated with a little **let** magic

# Lack of imagination?

*"Branding was, I think, a reasonable trade-off to make in 1983. I don't think that it's reasonable any longer."*

— Andrew Black

# Pre-Branding

All brands are themselves branded

There must be some initial 'pre-brand'

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

class preBrand.new → BrandInterface { ··· }
```

# Pre-Branding

The brand constructor puts it all together

```
let aBrand = preBrand.new
let Brand = aBrand.Type & BrandInterface

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

# AST Nodes

AST is used by the dialect type checkers

Many of the nodes have the same structural interface

```
let Decl = Node & type {
    name → String
    value → Expression
    pattern → Expression
}
```

Cannot safely use types to match against different node kinds

```
match(decl)
    case { varNode : Var → print "A var!" }
    case { defNode : Def → print "A def!" }
```

# AST Nodes

Branding the nodes provides distinct, nominal types

Depends on the implementation

- (Requires brands to be part of the standard language)

## Dialect Typing

Without brands, the node types are just run-time patterns

```
def Var = object {
    inherits pattern.abstract

    method match(o : Object) → MatchResult {
        Decl.match(o).andAlso { m.kind ≡ "var" }
    }
}
```

The type system doesn't know that this is a type

# Dialect Typing

Within a rule, the node is untyped

```
rule { varNode : Var →
    if (varNode.vallue.isEmpty) then { ··· }
}
```

If the dialect is built in the branding dialect, all of the node patterns can be treated as static types

# Exceptions

The exception hierarchy can now be implemented in Grace

```
class exceptionKind.name(name : String)
        brand(aKind : Brand) → ExceptionKind {

    method match(obj : Object) → MatchResult {
        aKind.Type.match(obj)
    }

    ...
}
```

# Exceptions

The exception hierarchy can now be implemented in Grace

```
class exceptionKind.name(name : String)
        brand(aKind : Brand) → ExceptionKind {

    method raise(message : String) → None {
        object is aKind { inherits exception; raise(message) }
    }

    ...
}
```

# Exceptions

The exception hierarchy can now be implemented in Grace

```
class exceptionKind.name(name : String)
        brand(aKind : Brand) → ExceptionKind {

    method refine(name : String) → ExceptionKind {
        exceptionKind.name(name) brand(aKind.extend)
    }

    ...
}
```

# Exceptions

The top of the hierarchy:

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

# Syntax

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

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

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

$$E ::= \tau \mid B \qquad \textit{(Static expression)}$$

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

# Well-formedness

Taking the type of any brand is well-formed

$$\frac{\overline{T} \vdash B \ \triangleright\ B'}{\overline{T} \vdash B.\mathsf{Type}\ \checkmark}$$

# Subtyping

Reflexivity just for named brands

$$\frac{}{\Sigma \vdash \beta.\mathsf{Type} <: \beta.\mathsf{Type}}$$

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

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

# Type Membership

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

# Typing

$$\frac{\cdot \vdash \textbf{type}\,\{\,\overline{S}\,\}\ \checkmark \qquad \Gamma, \mathsf{self} : \boxed{\mathsf{and}(\textbf{type}\,\{\,\overline{S}\,\}, \overline{B.\mathsf{Type}})} \vdash \overline{\textbf{method}\ S\ \{\, e\,\}}\ \checkmark}{\Gamma \vdash \textbf{object}\ \boxed{\textbf{is}\ \overline{B}}\ \{\,\overline{\textbf{method}\ S\ \{\, e\,\}}\,\} : \boxed{\mathsf{and}(\textbf{type}\,\{\,\overline{S}\,\}, \overline{B.\mathsf{Type}})}}$$

## Gradual Guarantee

Only permit runtime type testing on brand types?

Loses much of the ‘reified objects’ story