Tinygrace

A Simple, Safe, and Structurally Typed Language

Timothy Jones and James Noble
School of Engineering and Computer Science
Victoria University of Wellington, New Zealand
tim,kjx@ecs.vuw.ac.nz

ABSTRACT

Grace is a new gradually, structurally typed object-oriented programming language. Formal models of existing languages provide a rigorous base for claiming type soundness, so we have set about creating a model of a subset of Grace. While much of the formal literature of objects has used structural typing, models of popular modern languages such as Featherweight Java have had to use nominal typing to match the language they are modelling. In contrast to this, we present Tinygrace: a subset of Grace with a structural type system, feature-parity with FJ, and an accompanying proof of soundness.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages, Theory

Keywords

Grace, formal models, structural typing, language design

1. INTRODUCTION

The statically-typed languages of 'mainstream' object oriented programming almost exclusively use declarative classes to describe object structure, and nominal subtyping to describe their relationships. Modern formalisations of existing static OO languages have modelled predominantly class-based nominal systems as a result, despite much of the existing work utilising structural record typing [9]. The canonical example of a formalisation of an existing OO language is Featherweight Java [15], which strips away most of the features of Java to focus on the class interactions and typing, and provides a basis for future extension.

FTfJP '14, July 29 2014, Uppsala, Sweden

ACM 978-1-4503-2866-1/14/07 ...$15.00.
http://dx.doi.org/10.1145/2635631.2635848

In contrast to the classes and nominal systems of Java, *Grace* is a language that uses singleton object constructors and structural subtyping [5]. In order to gain similar benefits to the ones that FJ renders to Java, we have set about building a formalisation of a subset of the language. This paper presents *Tinygrace*, a model that strips Grace down to a bare, statically typed subset of Grace. We prove that Tinygrace follows a stricter definition of type soundness than FJ by preventing invalid casts, and claim feature-parity by informally discussing how each useful feature of FJ can be translated into Tinygrace.

2. AN OVERVIEW OF GRACE

The Grace programming language is a new object-oriented language with a simple classless object model [6]. Objects are built using singleton object constructor blocks, and declarations of methods and variables in the constructor are bound to the resulting object. Arbitrary statements may appear in the constructor as well, and are executed in the order written. In the following example we create a new object with a speak method, and announce the creation of the object:

```
object {
  method speak { print("My name is Miles") }
  print("Miles has been created")
}
```

Grace does include classes, but only as syntactic sugar for methods that build fresh objects. Classes do not introduce types [10], and objects retain no relationship with the class that created them. We can factor out the name of the object in the previous example by creating a class that creates speaking objects with a given name:

```
class person(name) {
  method speak { print("My name is {name}") }
  print("{name} has been created")
}
```

The person class is equivalent to a method with the same signature that wraps its body in an **object** constructor. The name parameter is available to the new object through the closure of the surrounding method. Note that braces in a string literal interpolate the contained expression into the resulting string.

Methods can be requested on an object using standard dot-syntax. An object created by the person class can be requested to speak with:

```
person("Nathan").speak
```

As there are no parameters, the parentheses delimiting arguments to the speak method are absent. The result of running the expression above is an announcement of Nathan's creation, and then Nathan's own introduction.

None of the previous examples feature types, as Grace is gradually typed [24]. Declarations which are not explicitly typed are given the type Unknown, which defers any checks on whether an object has a particular method until that method is requested on the object. This paper focuses on a statically typed subset of Grace, and so we provide explicit types in all the remaining examples.

Grace provides syntax for expressing the structural type of an object with a type literal that consists of a set of method signatures. These literals can be used directly as a type or bound to a name, which allows for mutual recursion. The type of the objects created in the previous examples could be declared as:

```
type Person = { speak → Done }
```

Every Grace method must return a value, and Done is the type of the 'unit' object done that is returned by methods that would normally have nothing to return, such as the print method. The type of an object with the person class in it could now be declared as:

```
type PersonFactory = {
  person(name : String) → Person
}
```

Grace has a single namespace that is shared by both types and values. Types are reified as objects at runtime and may be used freely in all of the same circumstances as values. The standard structural combinators & (intersection) and | (variant) are available both as operations on static types and as runtime operators. Rather than featuring an 'instance of' construct, reified Grace types have a method match that takes an object and returns whether that object's structure satisfies the conditions of the type. A match-case construct allows branching based on a series of type tests [14], avoiding invalid casts:

```
match (animal)
  case { p : Person → p.speak }
  case { d : Dog → d.bark }
  case { c : Cat → c.meow }
```

Objects may also inherit from methods with an **inherits** clause at the top of their constructor. The method that is being inherited from must return directly from another object constructor. The request in the clause subsumes the process of requesting the super constructor in class-based languages. As all Grace classes are just methods that immediately return an object, all classes can be inherited from.

```
object {
  inherits person("Marie")
  self.speak
}
```

Inheritance unifies the identities of the object constructors it builds in the inheritance chain, allowing standard class-based techniques such as up- and down-calls during initialisation, and ensuring that the object maintains a consistent identity. Unlike standard class inheritance, it does not establish any explicit typing relationship.

3. FORMAL MODEL

In this section we present Tinygrace, a formal model of a subset of the Grace language. Tinygrace is designed in the spirit of Featherweight Java [15], selecting the minimal set of features from Grace to reach feature parity with FJ. In order to preserve readability, Tinygrace programs are a sugared representation with simple transformations to an unsugared format. In the unsugared form, a Tinygrace program is just an expression, without any form of implicit information such as FJ's class table. Tinygrace has no dynamic typing and explicit type information is always required.

3.1 Syntax

The abstract syntax and sugar for Tinygrace is defined in Figure 1. The metavariable M ranges over methods, O over object constructors, B over case branches, x and y over variable names, X and Y over type names, τ over type expressions, S over method signatures, m over method names, and e over expressions.

We use $\overline{e}$ to indicate a possibly empty sequence of comma-separated expressions $e_1, \ldots, e_n$, as well as for method signature parameters $\overline{x : \tau}$, hiding parentheses when there are no parameters 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 $\overline{\textbf{case}\,\{\, x : \tau \rightarrow e \,\}}$ to indicate a non-empty sequence of case branches. Object constructors are the only value terms in Tinygrace.

In its sugared form, Tinygrace has classes, type declarations, inheritance, and a top-level type Object defined as **type** {}. Statements cannot be written in sequence — the top-level program must be a single expression. The desugaring process is a trivial transformation where all type declarations are substituted into each other, recursive references are captured, and classes are expanded into their method form. Inherits clauses must refer to a method outside of its direct scope that immediately returns an object, as in Grace. The clauses are desugared by copying down the methods in the super object, ignoring methods whose names are already defined in the sub-object, with the parameters of the request substituted into each copied method. Inheritance is used just to avoid repeating method definitions.

The sugar is not strictly necessary nor is it formally defined here, but it is useful for improving the readability of Tinygrace programs. Programs that cannot be desugared — either through recursive type references outside of a type literal or due to an unresolved inheritance request — are considered ill-formed.

Note that recursive type variables are only available over type literals, which prevents direct recursive references, and ensures that recursive type definitions are contractive [22]. A type declaration $\mu X.\textbf{type}\,\{\,\overline{S}\,\}$ can be written $\textbf{type}\,\{\,\overline{S}\,\}$ when X does not appear free in $\overline{S}$. The recursive type literal does not exist in Grace, but there is a trivial transformation between named type declarations and the recursive format.

Tinygrace uses Barendregt's variable convention [4, 26] that bound and free variables are distinct. This includes the variable self which is a keyword in Grace, but not in Tinygrace. Each object literal implicitly introduces a fresh variable self that is distinct from other definitions in the

Programs

$P ::= (\overline{T},\ \mathrm{e})$

Syntax

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

$O ::= \textsf{object}\ \{\, I\ \overline{M} \,\}$ *(Object constructor)*

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

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

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

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

Sugar

$T ::= \textsf{type}\ X = \tau \mid \textsf{type}\ X = \{\, \overline{S} \,\}$ *(Type alias)*

$C ::= \textsf{class}\ S\ \{\, I\ \overline{M} \,\}$ *(Class)*

$I ::= \textsf{inherits}\ \mathrm{m}(\overline{e})$ *(Inherits clause)*

Contexts

$\Pi ::= \cdot \mid \Pi,\ X$ *(Well-formed context)*

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

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

Auxiliary Definitions

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

Figure 1: Tinygrace grammar

$\boxed{\Pi \vdash \tau\ \checkmark}$

$$\frac{X \in \Pi}{\Pi \vdash X\ \checkmark}\ (\text{W-Name})$$

$$\frac{\overline{m}\ \textbf{distinct} \qquad \Pi, X \vdash \overline{\tau_p}, \tau_r\ \checkmark}{\Pi \vdash \mu X.\textsf{type}\ \{\, \overline{m(\overline{x : \tau_p}) \to \tau_r} \,\}\ \checkmark}\ (\text{W-Type})$$

$$\frac{\Pi \vdash \tau_1\ \checkmark \qquad \Pi \vdash \tau_2\ \checkmark}{\Pi \vdash \tau_1 \mid \tau_2\ \checkmark}\ (\text{W-}|)$$

$$\frac{\Pi \vdash \tau_1\ \checkmark \qquad \Pi \vdash \tau_2\ \checkmark}{\Pi \vdash \tau_1\ \&\ \tau_2\ \checkmark}\ (\text{W-}\&)$$

Figure 2: Tinygrace well-formedness judgements

surrounding scope.

Case branching in Tinygrace is the same as in Grace, though it is limited to just structural type judgments.

3.2 Static Semantics

The well-formedness judgements for Tinygrace are given in Figure 2. The rules are straightforward boilerplate, ensuring that type aliases exist, method names are distinct, and components of types are well-formed.

The rules for type and signature subtyping are given in Figure 3. 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 an assumption set Σ if, for every method in τ_2, there is a method with the same name in τ_1 with contravariant parameter subtypes and a covariant return subtype. Subtyping of recursive types follows a standard equirecursive definition [10, 21], where types are implicitly unfolded in Rule S-Lit.

The variant and intersection types have standard subtyping relationships. The rules for subtyping in the presence of intersection types are not complete, as they do not consider the two types $\textsf{type}\ \{\, m \to \tau_1 \,\}\ \&\ \textsf{type}\ \{\, m \to \tau_2 \,\}$ and $\textsf{type}\ \{\, m \to \tau_1\ \&\ \tau_2 \,\}$ to be subtypes even though their definitions are isomorphic. Resolving this issue requires expanding all intersections between structural types, an operation that produces $\bot$ when no object can inhabit both types at once. We do not address the issue for the sake of simplicity for the model.

The rules for typing of terms is given in Figure 4. The typing judgement for expressions has the form $\Gamma \vdash e : \tau$, meaning that in the variable environment Γ, the expression e has the type τ.

3.3 Dynamic Semantics

The rules for the reduction relation $\longmapsto$ are given in Figure 5. 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 reflexive and transitive closure of $\longmapsto$, which can 'run' a Tinygrace program e, resulting in either an object literal O or divergence.

Runtime type instance checking in Rule R-Inst is defined in terms of subtyping where an object's structural interface is compared to the given type. This resembles Rule 'R-Cast' from Featherweight Java [15], but without the re-

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

$$\frac{\tau_1 <: \tau_2 \in \Sigma}{\Sigma \vdash \tau_1 <: \tau_2}\ (\text{S-Assum}) \qquad \frac{\Sigma, \mathsf{type}\,\{\,\overline{S_1}\ \overline{S_1'}\,\} <: \mathsf{type}\,\{\,\overline{S_2}\,\} \vdash \overline{S_1 <: S_2}}{\Sigma \vdash \mathsf{type}\,\{\,\overline{S_1}\ \overline{S_1'}\,\} <: \mathsf{type}\,\{\,\overline{S_2}\,\}}\ (\text{S-Lit})$$

$$\frac{\Sigma \vdash \tau <: \tau_1}{\Sigma \vdash \tau <: \tau_1 \mid \tau_2}\ (\text{S-}|\text{-L}) \qquad \frac{\Sigma \vdash \tau <: \tau_2}{\Sigma \vdash \tau <: \tau_1 \mid \tau_2}\ (\text{S-}|\text{-R}) \qquad \frac{\Sigma \vdash \tau_1 <: \tau \qquad \Sigma \vdash \tau_2 <: \tau}{\Sigma \vdash \tau_1 \mid \tau_2 <: \tau}\ (\text{S-}|)$$

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

$\boxed{\Sigma \vdash S_1 <: S_2}$

$$\frac{\Sigma \vdash \overline{\tau_{p2} <: \tau_{p1}}, \tau_{r1} <: \tau_{r2}}{\Sigma \vdash m(\overline{x_1 : \tau_{p1}}) \to \tau_{r1} <: m(\overline{x_2 : \tau_{p2}}) \to \tau_{r2}}\ (\text{S-Sig})$$

Figure 3: Tinygrace subtyping judgement

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

$$\frac{\tau = \mathsf{type}\,\{\,\overline{m(\overline{x : \tau_p}) \to \tau_r}\,\} \qquad \cdot \vdash \tau\ \checkmark \qquad \Gamma, \mathsf{self} : \tau, \overline{\overline{x : \tau_p} \vdash e : \tau_r}}{\Gamma \vdash \mathsf{object}\,\{\,\overline{\mathsf{method}\ m(\overline{x : \tau_p}) \to \tau_r\ \{\,e\,\}}\,\} : \tau}\ (\text{T-Obj})$$

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

$$\frac{\Gamma \vdash e : \mathsf{or}(\tau_p) \qquad \Gamma, \overline{x : \tau_p \vdash e_c : \tau_c}}{\Gamma \vdash \mathsf{match}(e)\ \overline{\mathsf{case}\,\{\,x : \tau_p \to e_c\,\}} : \mathsf{or}(\overline{\tau_c})}\ (\text{T-Case}) \qquad \frac{\Gamma \vdash e : \tau_1 \qquad \cdot \vdash \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}\ (\text{T-Sub})$$

Figure 4: Tinygrace term typing judgement

$\boxed{e \longmapsto e'}$

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

$$\frac{\mathsf{method}\ m(\overline{x : \tau_p}) \to \tau_r\ \{\,e_r\,\} \in O_s}{O_s.m(\overline{O_p}) \longmapsto [O_s/\mathsf{self}, \overline{O_p/x}]e_r}\ (\text{R-Req})$$

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

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

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

$\boxed{O \in \tau}$

$$\frac{\cdot \vdash \mathsf{type}\,\{\,\overline{S}\,\} <: \tau}{\mathsf{object}\,\{\,\overline{\mathsf{method}\ S\ \{\,e\,\}}\,\} \in \tau}\ (\text{R-Inst})$$

Figure 5: Small-step semantics of Tinygrace reduction

quirement of an associated class table.

3.4 Properties

We demonstrate type safety through the standard theorems of progress and preservation [12, 18].

Theorem 1 (Progress). If $\cdot \vdash e : \tau$, then $e \longmapsto e'$ or there exists an O where $e = O$.

Progress is proved by induction on the derivation of $\cdot \vdash e : \tau$, with trivial cases for objects, variables, and subsumption; and straightforward induction for case expressions. Method requests perform a signature lookup in the type of the receiver, which is guaranteed to exist and have the appropriate parameter types by the premises of Rule T-Req, and then continues with the induction step.

Lemma 1 (Term Substitution Preserves Typing). If $\Gamma, x : \tau' \vdash e : \tau$ and $\Gamma \vdash O : \tau'$, then $\Gamma \vdash [O/x]e : \tau$.

Substitution preservation is proved through induction on the derivation of $\Gamma, x : \tau' \vdash e : \tau$. The cases are straightforward examinations of substitution.

Lemma 2 (Variant Subtraction). If $\cdot \vdash \tau_1 <: \tau_2 \mid \tau_3$ and $\cdot \vdash \tau_1 \not<: \tau_2$, then $\cdot \vdash \tau_1 <: \tau_3$.

This lemma is necessary to demonstrate that case branching is guaranteed to satisfy at least one of the branches, and is proved by induction of the derivation of $\cdot \vdash \tau_1 <: \tau_2 \mid \tau_3$, with trivial or contradictory steps for each rule.

Theorem 2 (Term Reduction Preserves Typing). If $\cdot \vdash e : \tau$ and $e \longmapsto e'$, then $\cdot \vdash e' : \tau$.

Preservation is proved by induction on the derivation of $e \longmapsto e'$. The congruence rules follow straightforward induction.

Case R-Req: $\mathsf{self} : \tau_s, \overline{x : \tau_p} \vdash e_r : \tau_r$ by the premises of the Rules T-Req and T-Obj, so $\tau = \tau_r$ by Lemma **1**.

Case R-Case: $O_m \in \tau_p$, and $\cdot \vdash O_m : \tau_p$ by the Rules R-Inst and T-Sub, so $\tau = \tau_c$ by Lemma **1**.

Case R-Miss: $\cdot \vdash O_m : \tau_m$, with $\cdot \vdash \tau_m <: \tau_p \mid \mathsf{or}(\overline{\tau_p})$ and $\cdot \vdash \tau_m \not<: \tau_p$, so by Lemma **2** there must be at least one more case (as $\mathsf{or}()$ is not defined). $\cdot \vdash e' : \mathsf{or}(\overline{\tau_c})$ by Rule T-Case, and removing the left variant does not affect the subtyping by Rule S-|-R, so $\cdot \vdash e' : \tau$ by Rule T-Sub.

Theorem 3 (Type Soundness). If an expression e is well-typed with $\cdot \vdash e : \tau$, and the reduction $e \longmapsto^* e'$ results in e' a normal form, then there exists O where $e' = O$ and $\cdot \vdash O : \tau'$ where $\cdot \vdash \tau' <: \tau$.

Type soundness follows from the two previous theorems.

Note that the only possible outcome of a completed Tinygrace program is an object literal — unlike FJ, which can get stuck on both invalid downcasts and 'stupid' casts.

3.5 Relationship with Featherweight Java

We have claimed that Tinygrace maintains feature-parity with Featherweight Java, in that all of the useful functionality of FJ can be replicated in Tinygrace in a type-safe manner. Here we outline each feature and discuss how it can be manually translated from FJ to Tinygrace.

The FJ class table can be represented in Tinygrace as an object, with each class adding a type definition and a complementary method inside the table object. Fields are implemented as methods that use the surrounding method closure to reference the parameter given to the constructing method. The expression component of an FJ program is translated into a method inside the table object, and that method is immediately requested. Constructor calls translate to requests for the methods on the table object, and casts become match-case branches.

Downcasts and stupid casts in an FJ program must have an alternate case manually supplied during the translation. Failing cases are necessary to reason about Java programs, but when we claim feature-parity with FJ's casting construct, we mean only that Tinygrace can recover dynamic type information lost in the static system (the intended purpose of downcasts). Stuck states are not considered useful.

The behaviour of a cast may also not have the same outcome between languages, as casts between two nominally distinct but structurally equivalent types will match objects created by either class in Tinygrace [17], but placing fresh, unused methods in types and their corresponding classes resolves the issue.

Consider the following truncated example from [15]:

```
class Pair extends Object { ... }
((Pair)new Pair(new Pair(new Object(),
    new Object())), new Object()).fst).snd
```

We can translate this into the following sugared Tinygrace:

```
type Pair = { fst → Object; snd → Object }
object {
  class Pair(fst : Object, snd : Object) → Pair {
    method fst → Object { fst }
    method snd → Object { snd }
  }
  method run {
    match (self.Pair(self.Pair(object {},
        object {}), object {}).fst)
      case { p : Pair → p.snd }
      case { o : Object → object {} }
  }
}.run
```

The example also illustrates the cost of ensuring safe casts: the downcast to Pair is clearly safe, but the type systems of both languages are insufficiently expressive to capture this fact, so an alternative case must be provided. This is a typical compromise required by type safety.

4. RELATED WORK

The popularity of nominal classes and types in statically-typed languages such as C++, Java, and C♯ have contributed significantly to the shift in focus of formal models, and this is cited by Bruce [9] as the reason for discussing predominantly nominal foundations of OO. Earlier foundational work such as Abadi and Cardelli's ς-family [2] focused entirely on structural object models.

Structural typing is becoming more popular in mainstream object-oriented programming languages, partly due to its compatibility with 'duck-typed' dynamic systems in existing languages. Modern efforts such as TypeScript [1] and older systems like Strongtalk [7, 8] use structural subtyping

to conservatively describe which methods and properties will be available on an object in existing dynamic languages. As these extensions add to existing dynamic languages, these efforts are also naturally gradually typed [25]. Scala supports structural types from Scala 2.6, but many formalisations of Scala's type systems, such as νObj [20] and FS_{alg} [11] are explicitly nominal, and do not address structural types. DOT [3] models Scala's structural types (among others), but does not prove type safety.

Experimentation of gradual typing with combinations of other features is of interest to future work on Tinygrace, as the full Grace language features generics [16] and may rely on type inference [23] to correctly type generic requests.

Other recent attempts at formalising new, structurally typed languages include Wyvern [19], an object-oriented language whose model translates to an extended lambda calculus. The design of the model uses a simple core language and iteratively extends it, translating each new language feature into an aspect of the previous language.

5. FUTURE WORK

Tinygrace represents the beginning of our formalisation of Grace proper and, like FJ, is intended for experimentation and extension. The two obvious immediate targets are the completeness of the type intersection combinator, which requires unsatisfiable types, and the full reification of types to objects. Other possibilities include adding generics for types and methods as in FGJ [15], and adding models of practical concerns such as references and heap allocation.

Modelling Grace's gradual typing is a larger extension, but necessary to begin representing the larger language. Introducing the Unknown type to the existing type system invalidates subtyping, as the type exists at every point on the type tree, and instead requires a notion of *consistent-subtyping* [24] that compares the 'known' portions of types. Losing some static checking also requires modelling Grace's dynamic contract system [14].

Our ultimate goal is to model Grace's dialect system [13], and provide a consistent notation for expressing new type rules as dialects, along with a transformation from this notation to actual implemented static checks.

6. CONCLUSION

In this paper we have presented Tinygrace, a formal subset of Grace, and proven soundness. Where many models of existing languages such as Featherweight Java formalise class-based and nominally typed systems, Tinygrace models singleton objects and structural subtyping instead.

The structural subtyping of Tinygrace allows the language to avoid the implicit class table of Featherweight Java, which FJ must carry in both a static and dynamic context. In Tinygrace, all information about a program is encoded directly into an expression. Despite this, Tinygrace maintains feature-parity with FJ and has stronger type safety, as it cannot enter a stuck state when recovering dynamic type information.

7. REFERENCES

[1] TypeScript. `http://www.typescriptlang.org`.

[2] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[3] N. Amin, A. Morrs, and M. Odersky. Dependent object types. In *FOOL*. ACM, 2012.

[4] H. Barendregt. *The Lambda Calculus*. North-Holland, revised edition, 1984.

[5] A. P. Black, K. B. Bruce, M. Homer, and J. Noble. Grace: the absence of (inessential) difficulty. In *Onward!* ACM, 2012.

[6] A. P. Black, K. B. Bruce, M. Homer, J. Noble, A. Ruskin, and R. Yannow. Seeking Grace: a new object-oriented language for novices. In *SIGCSE*. Springer, 2013.

[7] G. Bracha. The Strongtalk type system for Smalltalk. In *OOPSLA Workshop on Extending the Smalltalk Language*. ACM, 1996.

[8] G. Bracha and D. Griswold. Stongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA*. ACM, 1993.

[9] K. B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.

[10] W. R. Cook, W. Hill, and P. S. Canning. Inheritance is not subtyping. In *POPL*. ACM, 1990.

[11] V. Cremet, F. Garillot, S. Lenglet, and M. Odersky. A core calculus for Scala type checking. In *MFCS*. Springer, 2006.

[12] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.

[13] M. Homer, T. Jones, J. Noble, K. B. Bruce, and A. P. Black. Graceful dialects. In *ECOOP*, 2014. To appear.

[14] M. Homer, J. Noble, K. B. Bruce, A. P. Black, and D. J. Pearce. Patterns as objects in Grace. In *DLS*. ACM, 2012.

[15] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3), 2001.

[16] L. Ina and A. Igarashi. Gradual typing for generics. In *OOPSLA '11*. ACM, 2011.

[17] D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. In *ECOOP*. Springer, 2008.

[18] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM TOPLAS*, 10(3), July 1988.

[19] L. Nistor, D. Kurilova, S. Balzer, B. Chung, A. Potanin, and J. Aldrich. Wyvern: A simple, typed, and pure object-oriented language. In *MASPEGHI*. ACM, 2013.

[20] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *ECOOP*. Springer, 2003.

[21] D. J. Pearce. A calculus for constraint-based flow typing. In *FTfJP*. ACM, 2013.

[22] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[23] A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *POPL*. ACM, 2012.

[24] J. Siek and W. Taha. Gradual typing for objects. In *ECOOP*. Springer, 2007.

[25] N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. Bierman. Gradual typing embedded securely in JavaScript. In *POPL*. ACM, 2014.

[26] C. Urban, S. Berghofer, and M. Norrish. Barendregt's variable convention in rule inductions. In *CADE*. Springer, 2007.