Eliminating Distinctions of Class: Using Prototypes to Model Virtual Classes

DeLesley Hutchins
LFCS, University of Edinburgh
D.S.Hutchins@sms.ed.ac.uk

Abstract

In mainstream OO languages, inheritance can be used to add new methods, or to override existing methods. Virtual classes and feature oriented programming are techniques which extend the mechanism of inheritance so that it is possible to refine nested classes as well. These techniques are attractive for programming in the large, because inheritance becomes a tool for manipulating whole class hierarchies rather than individual classes. Nevertheless, it has proved difficult to design static type systems for virtual classes, because virtual classes introduce dependent types. The compile-time type of an expression may depend on the run-time values of objects in that expression.

We present a formal object calculus which implements virtual classes in a type-safe manner. Our type system uses a novel technique based on prototypes, which blur the distinction between compile-time and run-time. At run-time, prototypes act as objects, and they can be used in ordinary computations. At compile-time, they act as types. Prototypes are similar in power to dependent types, and subtyping is shown to be a form of partial evaluation. We prove that prototypes are type-safe but undecidable, and briefly outline a decidable semi-algorithm for dealing with them.

Categories and Subject Descriptors D.3.1 [*Software*]: Formal Definitions and Theory; F.3.2 [*Theory of Computation*]: Semantics of Programming Languages

General Terms Languages, Theory

Keywords Abstract Interpretation, Features, Dependent Types, Mixins, Partial Evaluation, Prototypes, Singleton Types, Virtual Classes, Virtual Types

1. Introduction

To a large degree, classes and inheritance are the "essence" of OO programming. A class encapsulates a group of interacting methods. Inheritance can be used to refine a class by adding new methods, or by overriding and extending existing ones. *Late binding* ensures that whenever a method is overridden, all references automatically point to the new version.

In a similar fashion, a module encapsulates a group of interacting classes. However, mainstream OO languages do not provide a mechanism for refining modules. C++ namespaces and Java packages are modules, and nested or inner classes can also be used to emulate modules to some degree. However, none of these mechanisms support late binding. It is possible to add new classes to a module, but it is not possible to refine or replace an existing class definition with something else.

As has been extensively argued elsewhere [14] [48] [26], it is not possible to refine a group of mutually recursive classes by deriving a new, parallel set with different names. The act of renaming the classes breaks the type dependencies between them, thus requiring downcasts and run-time type checks. The situation is somewhat analogous to the Java collection classes before generics were introduced; downcasts are necessary because there is no way to express the appropriate type relationships.

This issue is illustrated by the "expression problem", which has been discussed extensively in the literature [44] [27] [50] [37] [55]. The expression problem concerns the design of an extensible interpreter or compiler. A compiler manipulates abstract syntax trees, which consist of several different kinds of node — e.g. literals, operators, declarations, etc. A compiler also defines several operations over such trees — e.g. evaluation, pretty-printing, typing, etc. A compiler extension must add a new kind of node, or a new operation, without altering any definitions in the original source code.

In a functional programming language it is easy to extend the compiler with new operations, but hard to add new kinds of node. In a OO language it is easy to add new kinds of node, but hard to add new operations, as is illustrated by the following example in Java:

```
// core definitions
abstract class Expr { };

class Literal extends Expr {
  int value;
};

class Plus extends Expr {
  Expr left;
  Expr right;
};

// extensions
interface ExprEval {
  int eval();
};

class LitEval extends Literal
              implements ExprEval {
  int eval() { return value; }
};

class PlusEval extends Plus
               implements ExprEval {
```


OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.

```
  int eval() {
    return ((ExprEval) left).eval() +
           ((ExprEval) right).eval();
  }
};
```

This example follows the interpreter design pattern, where each kind of node has a different class, and each operation is a method that must be implemented by all the classes. Since the original classes cannot be modified, this code defines a new operation within an interface. It then extends each of the node classes to implement the interface.

The trouble with this approach is that left and right have type Expr, not ExprEval. Whenever we wish to call one of the new operations, we must insert a downcast to the appropriate type. The situation is somewhat analogous to the Java collection classes before generics were introduced; downcasts are necessary because there is no way to express the appropriate type relationships.

One way to solve the expression problem in an elegant fashion is to use *virtual classes* [26]. The following example is pseudo-code for a Java-like language. It encapsulates the entire group of classes as a module, and then refines them simultaneously by refining the module as a whole [28]. This mechanism allows us to add a method to each class in the group, without altering the the original module. Type dependencies within the module are preserved because classes keep the same name.

```
module EBase {
  abstract class Expr { };

  class Literal extends Expr {
    int value;
  };

  class Plus extends Expr {
    Expr left;
    Expr right;
  };
};

module EvalMod extends EBase {
  refine class Expr {
    int eval();
  };

  refine class Literal {
    int eval() { return value; }
  };

  refine class Plus {
    int eval() {
      return left.eval() + right.eval();
    }
  };
};
```

Virtual classes are defined by analogy with virtual methods. Just as it is possible for a derived class to override a virtual method, it is possible for a derived module to *refine* a virtual class. Class refinement obeys the same rules as ordinary inheritance; the new classes inherit definitions from the old ones.

Class names in Java are global identifiers. As a result, the relationships between classes are hard-coded, and class hierarchies become brittle and difficult to extend. With virtual classes, class names are local references within the current module, and inheritance between modules preserves the relationships between classes. EvalMod.Literal extends EvalMod.Expr, even though it is not explicitly declared, because EBase.Literal extends EBase.Expr, and EvalMod inherits from EBase.

In a fully general implementation of this concept, modules can also contain virtual nested modules. Module inheritance thus has the potential to be truly scalable, since it can be used to manipulate structures of arbitrary size. If a class library is encapsulated in a module, then it is possible not just to link against the library, but to customize and refine it to the needs of a particular application.

1.1 History

Virtual classes were first introduced in the Beta language, and were initially presented as an alternative to parameterized classes — a.k.a. generics [39]. However, Beta was unable to do static typing for virtual classes, and was forced to insert run-time checks.

The Beta language also had a major restriction, which is that it was not possible to inherit from a virtual class. This rules out solutions like the one above, which is an example of a *higher-order hierarchy* [27]. The gbeta language removed this restriction, and provided a static type system [24]. However, the type system for gbeta is complex, and has only recently been formalized [28].

Feature-oriented programming [8] [7], and multi-dimensional separation of concerns [44] are a different approach to the same problem. These solutions provide a mechanism for module refinement similar to the one shown above, but they do so by transforming code. Source code transformation happens outside the type system of the target language, so features cannot be type-checked separately.

1.1.1 Recent work

Interest in virtual classes has resurged in the past few years, and a number of formal models have recently been published in the literature; see section 8 for more details. The biggest difference between them lies in the way they treat modules. There are three main approaches:

1. Modules are distinct entities, often called “class groups”.
2. Modules are classes. Virtual classes are static nested classes.
3. Modules are objects. Virtual classes are inner classes.

Using a distinct notion of class group makes static typing easier, but is somewhat unsatisfying because it ignores the similarity between module inheritance and class inheritance. The second two approaches present a unified model of inheritance, but differ in whether they regard virtual classes as attributes of classes, or attributes of objects.

Treating virtual classes as attributes of objects is the most powerful approach, because it allows an arbitrary number of distinct class families to be created at run-time. However, treating virtual classes as attributes of classes is often more convenient, because there is no need to pass around instances of the enclosing classes.

We treat virtual classes as attributes of objects, which is the most difficult case to model formally. However, our solution uses a prototype model in which classes *are* objects, so it encompasses both of the latter approaches.

1.2 Challenges

Formal models of virtual classes have proved difficult to build because classes are complex entities, which serve three distinct roles in OO programs:

1. A class C is a generator for instances of the class.
   — e.g. new C(...)
2. A class C is a generator for subclasses.
   — e.g. class D extends C { ... }
3. Classes denote types.

The first role presents no difficulties; virtual classes behave much like virtual methods. The expression new myobj.C will locate the constructor for C at run-time, by looking it up in the virtual

method table of myobj. To ensure that this operation is safe, refinements of a virtual class must share the same constructor signatures.

The second role is harder, because refinements to a virtual base class will affect any derived classes. Some refinements may even invalidate derived classes. For example, assume that we have two virtual classes, C and D, where C declares a virtual method m, and D overrides m. If C is later refined such that m becomes final, then that refinement will invalidate D. The solution we present in this paper detects such errors by re-checking the interface of all inherited definitions.

The third role is the most difficult. In a method call to myobj.m(), the code for m() is not determined until run-time, because it depends on the dynamic value of myobj. Virtual classes behave similarly; the class denoted by the expression myobj.C depends on the dynamic value of myobj.

If virtual classes are attributes of objects, then the type system must necessarily involve some form of *dependent types* — types that depend on objects [38]. Dependent types are tricky because static safety requires that type judgments be performed at compile-time, even though the precise value of myobj may not be known until run-time.

1.2.1 Recursive Modules

The situation is further complicated by the pervasive presence of recursion. Each class in a module can potentially refer to all the others. The interface of a module is thus a set of mutually recursive type equations. Similarly, all of the methods and constructors in a module can potentially call each other. The implementation of a module is thus a set of mutually recursive definitions.

Inheritance relationships add an additional twist, because inheritance is *not* recursive. The inheritance graph cannot have any cycles, as would occur if classes A and B tried to inherit from each other. Nor should there be any infinite descending chains, as might occur if a nested class tried to inherit from its enclosing class.

Type theories for recursive modules are not simple. The traditional approach is to untangle recursive types from recursive implementations [22], but such untangling does not seem possible in the presence of dependent types. The alternative is to permit general recursion at the type level, but that yields a theory that is undecidable. Our approach is to accept undecidability as a necessary evil, but to develop a practical and decidable semi-algorithm for type checking.

1.3 Summary of Paper

We present the DEEP calculus, which provides type-safe support for virtual classes. Its semantics is based on a model of inheritance much like that in gbeta, which Ernst refers to as *propagating combination* [25]. We prefer the term used by Odersky and Zenger: *deep mixin composition* [55].

The most technically innovative aspect of DEEP is the fact that the type system is based on *prototypes* rather than types. Prototypes are first-class objects; they exist at run-time, and they can be used in ordinary expressions. Prototypes are also types, however, and there is a subtype relation defined between them. Static type safety relies on subtyping, rather than typing.

Prototypes allow us to treat modules, classes, methods, and objects in a uniform manner, as entities which exist at both compile-time and run-time. As such, they provide a natural mechanism for dealing with dependent types. In a dependent type system, typing may involve the evaluation of object expressions. In our prototype model, this is reflected in the fact that the subtype relation has a copy of evaluation embedded in it. In fact, subtyping can be seen as a mixture of partial evaluation and abstract interpretation.

Subtyping in DEEP is both nominal and structural. It is defined as a structural relation over terms, but terms can be paths of the form *object_name.class_name*, which means that nominal subtyping is included as a special case.

We show how standard approaches to type soundness can be adapted to prototypes, and we prove that the type system for DEEP is sound. We also show that the type system is undecidable, and provide an informal discussion of techniques to overcome this limitation.

Despite the theoretical nature of the topic, our goal for this paper is not to be overly technical. Rather, we hope to present enough theory to satisfy critics that there is some basis to our claims, while providing a discussion that is readable by a wider audience.

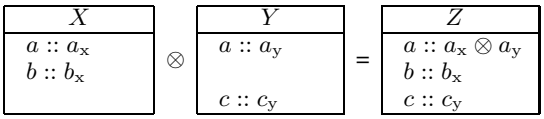
The organization of this paper is as follows. Section 2 provides an informal discussion of issues that motivated the design of DEEP. Section 3 introduces the DEEP calculus. Section 4 gives some example code. Section 5 describes subtyping, and section 6 covers type safety and decidability. Section 7 describes extensions to the calculus. Section 8 covers related work, and section 9 concludes.

2. Informal Discussion

Inheritance hierarchies are usually presented as a directed graph, where the nodes are classes and the arcs are inheritance relationships. In keeping with this idea, method lookup is frequently defined as a graph traversal. Unfortunately, this metaphor becomes unwieldy when there are inheritance relationships between the modules themselves as well as the classes within those modules.

Feature-oriented programming provides a compositional approach to modeling inheritance [8]. Features are essentially partial definitions of modules, and complex features are created by composing simpler ones. A compositional approach is convenient for a formal calculus, so DEEP is based on feature composition rather than inheritance.

A *deep mixin composition* of modules is one which descends into the module hierarchy, and recursively composes classes, methods, and nested modules which have the same name. It is defined as follows, where "$\otimes$" is the composition operator:



In this example, X and Y are two modules, while a, b, and c are named definitions, which may be modules, classes, methods, or constants. X and Y each declare a partial definition, or fragment of a. When X and Y are composed together, the composition operator is recursively applied to compose the two fragments. Definitions which exist in one module but not the other are copied over unchanged.

Class composition is the same as module composition; the resulting class "inherits" methods from both sources, and recursively composes methods with the same name. The composition of two methods or constants must merge their implementations in some way, as will be discussed below.

The "::" syntax is DEEP notation for an open or virtual binding. It means that a must contain at least the definitions in a_x, and at least those in a_y. If a is a class, then it must be a subclass of both a_x and a_y.

2.1 Symmetric Composition

DEEP provides two distinct operators for performing deep mixin composition. The first operator, written "&", stands for symmetric composition, which is similar to the way *traits* are composed [45]. Symmetric composition obeys the following algebraic identities:

$$
\begin{array}{rcll}
t \,\&\, t & \equiv & t & \text{Idempotence} \\
t \,\&\, u & \equiv & u \,\&\, t & \text{Commutativity} \\
(t \,\&\, u) \,\&\, s & \equiv & t \,\&\, (u \,\&\, s) & \text{Associativity}
\end{array}
$$

These three identities constitute the algebraic definition of a semi-lattice, which means that there is a partial order defined over modules as follows:

$$t <: u \text{ if and only if } t \,\&\, u \equiv t \quad \text{(Definition of Subtyping)}$$

Subtyping is defined directly over modules, rather than types. The intuition is that every module provides a set of capabilities. A module t is a subtype of u if t provides all of the capabilities that u provides. Due to its relationship with subtyping, the "&" operator can be viewed as a form of *type intersection*.

Unlike traditional type systems, the "capabilities" of a module encompass not only its interface, but also its implementation. Two modules which have the same interface, but provide different implementations, are regarded as distinct entities; neither of two is a subtype of the other.

Unfortunately, type intersection is not powerful enough to model OO inheritance. Commutativity and idempotence are strong requirements, and while it is easy to compose interfaces in this manner, it does not seem possible to compose method implementations. If two classes in an intersection both define a method with the same name, then one definition must be abstract — it cannot have a body.

2.2 Asymmetric Composition

The "&*" operator is used for asymmetric composition, which can be used to model features, mixin layers[46], and OO inheritance. Asymmetric composition is identical to type intersection, except that it also merges implementations. When two classes both implement a method with the same name, then the bodies of the two methods will be glued together in some way.

The difficulty of merging implementations from two sources is well-known in OO programming as "the multiple inheritance problem". The simplest way to resolve conflicts is by automatic overriding — definitions on the right-hand side of the composition override those on the left. This does not merge two implementations, it simply replaces one with the other.

A more sophisticated approach is to use *mixin composition* [11] [10] [29]. Definitions on the right-hand side are functions which *transform* definitions on the left. Mixin composition is a generalization of the way overriding works in OO inheritance, where a subclass may refer to definitions in the superclass with the **super** keyword. Mixins are a powerful technique, but that power comes at a cost:

Ordering Problems: Since mixin composition is not commutative, the order in which modules are composed is significant. Rearranging the order will generate different behaviors, with effects that may range from subtle to catastrophic.

Duplicates: Since mixin composition is not idempotent, it is possible to apply the same transformation twice, as illustrated in the following example:

$$
\begin{array}{l}
\text{let } t = a \mathbin{\&*} b \\
\text{let } u = a \mathbin{\&*} c \\
\text{let } s = t \mathbin{\&*} u \quad (= a \mathbin{\&*} b \mathbin{\&*} a \mathbin{\&*} c)
\end{array}
$$

This is merely a restatement of the classic "diamond" problem with multiple inheritance. The implementation of a occurs twice within s, which may or may not be what was intended.

A number of OO languages, including CLOS, Dylan, and Python, *linearize* the multiple inheritance hierarchy [6]. Linearization eliminates duplicates, which is perceived as the greater of the two evils, but it does not solve the ordering problem.

The DEEP calculus does not perform either mixin composition or linearization by default. The core calculus is defined with simple overriding semantics. However, this mechanism can be extended to more sophisticated kinds of composition without altering any of the type judgments presented in this paper.

2.3 Interfaces

Asymmetric composition differs from type intersection only where implementations are concerned, which means that the two operators are identical with respect to *interfaces*.

Every module t has a unique interface $\uparrow t$, such that $t <: \uparrow t$. Like interfaces in Java, the interface of a module only contains type signatures for methods and fields; all implementations are stripped away. Interfaces obey the following identity:

$$\uparrow(f \mathbin{\&*} g) \equiv \uparrow f \,\&\, \uparrow g$$

Interfaces in DEEP serve much the same role as types in other languages

2.4 Virtual Types and Dependent Types

A type which is an attribute of an object is known as a *virtual type*. Static type safety for virtual types is illustrated by the following example, which is adapted from [49]. Consider a program which must deal with several different kinds of animal, each of which eats a different kind of food. We model this by using a virtual type which holds the kind of food that the animal eats, as illustrated by the following psuedo-code:

```
class Food { ... };
class Meat extends Food { ... };

abstract class Animal {
  type FoodType <: Food;  // virtual type

  abstract FoodType favoriteFood();
  abstract void     eat(FoodType food);
};

static void feed1(Animal aml) {
  aml.eat(new Meat());           // error!
}

static void feed2(Animal aml) {
  aml.eat(aml.favoriteFood());   // OK
}
```

A simplistic and unsound approach is to assume that since aml is an animal, and animals eat food, then it is possible to feed any kind of food to aml. This is incorrect because it would allow us to feed meat to cows, or grass to people. Nevertheless, it is hard to come up with a good alternative, because if aml is some arbitrary animal, then there is no way of knowing at compile-time what kind of food it eats!

The code above circumvents this problem by endowing every animal with a favoriteFood which is guaranteed to be edible, even though its exact type is not statically known. The expression aml.favoriteFood() has type aml.FoodType. The argument to aml.eat must also have type aml.FoodType, so function feed2 is well-typed.

The type aml.FoodType is a *dependent type*, because the object aml appears within the type expression. Two types $a.T$ and $b.T$ are the same only if a and b refer to the same object. In this case, object identity follows because the object aml is referred to by name.

2.5 Object equality and evaluation

In a dependent type system, proving that two types are equivalent may involve proving that two object expressions are equivalent, and there is no way prove such equivalence in general. Every dependent

type system must make a design decision about how to deal with object equality.

Array types in C are a primitive form of dependent type, because the size of the array is an integer expression. Most modern compilers can show that the type int[256] is the same as int[4*64], because they evaluate constant integer expressions at compile-time. However, array sizes in C must be constant expressions involving built-in operators, which places a limit on what the type system is expected to prove.

Like C, the DEEP calculus uses an intensional notion of equality which is based on evaluation. Two expressions are equal if they can be reduced to a common term. Unlike C, the type-checker may reduce arbitrary expressions at compile-time, not just constant ones. The type system for DEEP can prove that $(\lambda x.x)(y) = y$, because the left-hand side reduces to the right, even if y is not statically known. However, it cannot prove that $x + y = y + x$, where x and y are unknown variables; DEEP does not handle arithmetic identities.

Because it may evaluate arbitrary expressions at compile-time, the type system for DEEP is structured much like an online partial evaluation engine [35]. In fact, we use a technique similar to partial evaluation not only to prove type equivalence, but also to derive the types themselves. This technique is based on the idea of prototypes.

2.6 Prototypes

Prototypes were proposed in the Self language as an alternative to classes [51]. In a prototype-based system, objects are self-descriptive. A *prototypical object* is one which implements some default behavior. New objects are created by refining existing prototypes. Prototype refinement differs from inheritance mainly because it is an operation over objects rather than classes, so it can occur at run-time.

The DEEP calculus is a prototype-based system in this sense. Classes and modules are treated as ordinary objects which exist at run-time, and new classes are created by refining or composing existing ones.

The DEEP calculus is also prototype-based in a different sense: prototypes in DEEP can act as both types and objects. Unlike Self, DEEP is statically typed. There is a subtype relation defined over all objects, and composition creates subtypes.

Objects in DEEP may be only partially defined. An *interface* is an object which is fully abstract; it contains only type signatures for methods and fields. An *instance* is an object which is fully concrete; every method has an implementation, and every field has a value. A class is somewhere in between — it defines implementations for methods, but it may not define values for fields. Throughout the rest of this paper, we will refer to interface prototypes as "types", and concrete prototypes as "objects", but the calculus itself does not distinguish between the two.

Composition in DEEP is used to create both subtypes and instances. An instance of a type is a subtype of that type, which means that instances are leaves at the bottom of the inheritance hierarchy. For example:

$$\begin{array}{cccccccc} 3 & <: & \mathsf{Int} & <: & \mathsf{Number} & <: & \mathsf{Object} \\ \mathsf{true} & <: & \mathsf{Bool} & <: & \mathsf{Object} \end{array}$$

In DEEP, all objects can act as types. A concrete value, like the number 3, is a *singleton type* [47]; it represents the type of all integers which are equal to 3.

In addition to treating objects as first-class types, DEEP treats types as first-class objects. The result of performing an operation on a type is just another type, e.g.

$$\begin{array}{cccc} 1 + 2 & = & 3 & \qquad\qquad \mathsf{true}\ \&\&\ \mathsf{Bool} = \mathsf{Bool} \\ \mathsf{Int} + 2 & = & \mathsf{Int} & \qquad\qquad \mathsf{false}\ \&\&\ \mathsf{Bool} = \mathsf{false} \end{array}$$

The intuition behind this mechanism is that interface types such as Int represent objects which are only partially defined. It is possible to perform computations with such objects, but the result may also be partially defined.

The ability to perform computations with prototypes provides a link between evaluation and typing. The bounding type of an expression can be calculated by substituting types for any unknown variables, and then evaluating the expression. In other words, typing is a form of *abstract interpretation* [21].

3. Syntax and Semantics

The syntax and operational semantics of DEEP are given in figure 1. There are two basic constructs: functions and records. Functions are defined much the same as in other languages. Objects, classes, and modules are all encoded as records.

For the sake of clarity, the following discussion assumes that the core calculus has been extended with types Int and Bool, integer and boolean literals such as 0 and true, as well as the standard logical and arithmetic operators. These objects are not built-in; they can be encoded using more primitive constructs.

3.1 Functions

A function is defined by the syntax $\lambda(x : t).\ u$. The variable x is the argument, t is the argument type, and u is the body. As is standard practice, functions with multiple arguments are encoded by Currying:

$$\lambda(x, y, z).\ u \quad \text{is short for} \quad \lambda(x).\ \lambda(y).\ \lambda(z).\ u$$

3.2 Records

A record is defined by the syntax $\mu x\{\overline{d}\}$, where $\overline{d}$ is a (possibly empty) sequence $d_1\ ..\ d_n$ of labeled declarations, called *slots*. A record may not have more than one slot with the same label. The self-variable x is an identifier which refers to the enclosing record; it is equivalent to the **this** keyword in C++ or Java [1]. (Using an explicit name for **this** is helpful when dealing with nested records.)

A slot is projected from a record using standard OO dot notation, as shown below. One slot can refer to other slots within the same record by means of the self-variable x.

```
let r = μx{ a: Int = 1;
            b: Int = 2;
            c = x.a + x.b;
          }

r.c ⟶ r.a + r.b ⟶» 1 + 2 ⟶ 3
```

Slots use late binding. In the example above, x is not assigned a value until the slot c is actually projected from the record. The definition of c thus acts like a simple method, rather than a constant. If the values of a and b were overridden by composition, then the value of c would be affected accordingly.

Every record denotes a fixpoint. This fixpoint can be used to declare recursive objects, including mutually recursive functions and circular data structures, just like the letrec construct in functional languages. Late binding for slots has the same semantics as lazy evaluation in a letrec.

The slots of a record hold prototypes, which can be either types or objects. This means that records can also be used to declare recursive *types*, such as lists and trees.

3.3 Declarations

Declarations serve two distinct roles. First, a declaration defines an object, which is stored in the slot of a record. Second, a declaration establishes a constraint which must hold in subtypes of that record. This dual role — object + constraint — is what allows records

x, y, z	Variable	$\otimes$ ::=	Composition:
ℓ, l, m	Slot label	$\&$	type intersection
		$\&*$	asymmetric composition
s, t, u ::=	Terms:		
x	variable	v, w ::=	Normal Forms
$\lambda(x : t).\, t$	function	$\lambda(x : t).\, t$	function
$\mu x\{\overline{d}\}$	record	$\mu x\{\overline{d}\}$	record
$t(t)$	application		
$t.l$	projection	Γ ::=	Contexts
$t \otimes t$	composition	$\emptyset$	empty context
$\uparrow t$	interface	$\Gamma, x \lhd t$	variable declaration
c, d, e ::=	Declarations:	$\lhd$ ::=	Type relations
$l :: t$	open/virtual binding	$<:$	subtype
$l = t$	final binding	$\lessdot$	exact subtype
$l : t = u$	concrete field	$\equiv$	type equivalence
$l : t = \_$	abstract field		

Notation:

- $\overline{d}$ denotes a possibly empty sequence of declarations d_1 .. d_n, where each declaration is terminated by a semicolon.
- $\mathrm{dom}(\overline{d})$ denotes the set of labels in the sequence $\overline{d}$.
- d_ℓ denotes the declaration with label ℓ in the sequence $\overline{d}$.
- $[x \mapsto t]\, u$ denotes the capture-avoiding substitution of the term t for the variable x within u.
- $\overline{c} \otimes \overline{d}$ is defined below. It concatenates the two sequences, composing declarations which have the same label. $\overline{c} \otimes \overline{d} \longrightarrow \overline{e}$ and $\overline{c} \otimes \overline{d} \Rrightarrow \overline{e}$ will compose declarations using "$\longrightarrow$" and "$\Rrightarrow$", respectively. (See figure 2)

Term Equality: α-renaming of bound variables x, plus $\dfrac{\overline{d} \text{ is a permutation of } \overline{e}}{\mu x\{\overline{d}\} = \mu x\{\overline{e}\}}$

Evaluation Context: $E ::= [] \mid E.l \mid E(t) \mid v(E) \mid E \otimes t \mid v \otimes E$

Reduction: $\boxed{t \longrightarrow t}$

$$\frac{t \longrightarrow t'}{E[t] \longrightarrow E[t']} \quad \text{(E-CONGRUENCE)}$$

$$(\lambda(x : t).\, u)(v) \longrightarrow [x \mapsto v]\, u \quad \text{(E-APPLY)}$$

$$\frac{\mathrm{declVal}(d_\ell, t)}{\mu x\{\overline{d}\}.\ell \longrightarrow [x \mapsto \mu x\{\overline{d}\}]\, t} \quad \text{(E-PROJECT)}$$

$$\begin{array}{ll} \mathrm{declVal}(l :: t, & t) \\ \mathrm{declVal}(l = t, & t) \\ \mathrm{declVal}(l : t = \_, & t) \\ \mathrm{declVal}(l : t = u, & u) \end{array}$$

$$\lambda(x : t).\, u \;\otimes\; \lambda(x : t').\, s \;\longrightarrow\; \lambda(x : t').\, (u \otimes s) \quad \text{(E-COMPOSEFUN)}$$

$$\frac{\overline{c} \otimes \overline{d} \longrightarrow \overline{e}}{\mu x\{\overline{c}\} \;\otimes\; \mu x\{\overline{d}\} \longrightarrow \mu x\{\overline{e}\}} \quad \text{(E-COMPOSEREC)}$$

$$\uparrow\lambda(x : t).\, u \;\longrightarrow \lambda(x : t).\, \uparrow u \quad \text{(E-INTERFACEFUN)}$$

$$\uparrow\mu x\{\overline{d}\} \longrightarrow \mu x\{\uparrow\overline{d}\} \quad \text{(E-INTERFACEREC)}$$

Composition of sequences: $\boxed{\overline{c} \otimes \overline{d}}$

$$\overline{c} \otimes \overline{d} \stackrel{\text{def}}{=} \overline{e}, \text{ where:}$$

$$\mathrm{dom}(\overline{e}) = \mathrm{dom}(\overline{c}) \cup \mathrm{dom}(\overline{d})$$

$$e_\ell = \begin{cases} c_\ell \otimes d_\ell & \text{if } \ell \in \mathrm{dom}(\overline{c}) \cap \mathrm{dom}(\overline{d}) \\ c_\ell & \text{if } \ell \in \mathrm{dom}(\overline{c}) \text{ and } \ell \notin \mathrm{dom}(\overline{d}) \\ d_\ell & \text{if } \ell \in \mathrm{dom}(\overline{d}) \text{ and } \ell \notin \mathrm{dom}(\overline{c}) \end{cases}$$

Composition of declarations: $\boxed{c \otimes d \longrightarrow e}$

$$(l :: t) \otimes (l :: u) \longrightarrow l :: t \otimes u \quad \text{(DE-COMPOSE)}$$

$$\frac{e \text{ replaces } d}{d \otimes e \longrightarrow e} \qquad \frac{d \text{ overrides } e}{d \otimes e \longrightarrow d} \quad \text{(DE-REPLACE)}$$

$$\begin{array}{lll} (l = t) & \text{replaces} & (l = t') \\ (l : t = \_) & \text{replaces} & (l : t' = \_) \\ (l : t = u) & \text{replaces} & (l : t' = u') \\ (l = t) & \text{overrides} & (l :: u) \\ (l : t = u) & \text{overrides} & (l : t' = \_) \end{array} \qquad \frac{d \text{ overrides } e}{d \text{ replaces } e}$$

Interface of declarations: $\boxed{\uparrow d \stackrel{\text{def}}{=} e}$

$$\begin{array}{lll} \uparrow(l :: t) & \stackrel{\text{def}}{=} & l :: \uparrow t \\ \uparrow(l = t) & \stackrel{\text{def}}{=} & l = t \\ \uparrow(l : t = \_) & \stackrel{\text{def}}{=} & l : t = \_ \\ \uparrow(l : t = u) & \stackrel{\text{def}}{=} & l : t = \_ \end{array} \quad \text{(DE-INTERFACE1-4)}$$

Figure 1. Syntax and Operational Semantics

and functions to serve as prototypes. There are three main kinds of declaration, which establish three different constraints:

- $l :: t$ is an *open* or *virtual* declaration, which has an upper bound given by t. An open declaration can be specialized to any subtype of t. Virtual methods and virtual classes are defined using open declarations.
- $l = t$ is a *final binding*. A final binding cannot be specialized; all subtypes must have equivalent definitions.
- $l : t = u$ is a field. The term t is the *range* of the field, while u is the (optional) *implementation*, which must be a subtype of the range. The implementation may be left unspecified using the syntax $l : t = \_$, in which case the field is *abstract*. Fields with implementations are *concrete*.

The range of a field is invariant; all subtypes must have the same range. The implementation of a field can be overridden in a composition, as will be discussed below.

3.4 Composition

Composition is defined over both terms and declarations. The composition of two records will compose declarations with the same label. Declarations are composed as follows. (The "$\otimes$" symbol is a meta-variable which ranges over both "&" and "&*".)

- If both declarations have open bindings, then composition is recursively applied to terms:
  $(l :: t) \otimes (l :: u) \longrightarrow (l :: t \otimes u)$
- If one declaration is more specific than the other, then the specific one overrides the general one. Final bindings override open bindings, and concrete fields override abstract fields. If two declarations are equivalent, then the one on the right replaces the one on the left.
- When two fields are composed with &*,
  (e.g. $(l : t = u)$ &* $(l : t = u')$) then the implementation on the right replaces the implementation on the left.

The composition of two functions composes their bodies:

$$\lambda(x : t).\, u \;\otimes\; \lambda(x : t).\, s \;\longrightarrow\; \lambda(x : t).\, u \otimes s$$

Note: in the operational semantics, "&" and "&*" are treated identically. The difference between the two only appears in the type system. The expression t & u is not well-formed unless there is a valid intersection of the two terms.

3.5 Interfaces

The main purpose of a field is to separate the interface of a record from its implementation. The expression $\uparrow t$ will erase the implementation of all fields. Like "$\otimes$", "$\uparrow$" is recursively defined over both terms and declarations. It will erase the implementation of every field that could potentially be overridden by "&*". For example:

```
↑μx{ m :: μy{                    μx{ m :: μy{
       a: Int = 1;                      a: Int = _;
     };                    ⟶         };
     b: Int = 2;                      b: Int = _;
   }                                }
```

3.6 Function types

In keeping with the prototype model, functions can act as types. Subtyping between functions is defined pointwise: given two functions f and g, $f <: g$ implies that $f(a) <: g(a)$, for all a. For example:

$$\lambda(x : \mathsf{Int}).\, x + x \quad <: \quad \lambda(x : \mathsf{Int}).\mathsf{Int}$$

Two functions are compared by comparing their bodies, treating the variable x as a free variable with the given upper bound. The example above holds because $x + x$ is a subtype of Int if x is a subtype of Int. The prototype $\lambda(x : \mathsf{Int}).\mathsf{Int}$ serves the same role as the conventional arrow-type $\mathsf{Int} \to \mathsf{Int}$.

The argument type of a function is invariant (rather than contravariant) in subtypes. This is because the composition of two functions will mix their bodies together. Since the body of a function may refer to its argument, relaxing the type constraint would not be type-safe.

3.7 Record types

Subtyping between records is done by comparing individual slots. Final bindings and fields specialize open bindings, and concrete fields specialize abstract fields. E.g.

```
μx {                              μx {
  a :: Int;                         a :: Object;
  b = 0;                            b :: Int;
  c = 1;              <:            c = 1;
  d : Int = 2;                      d : Int = _;
  e = false;                      }
}
```

When comparing two records, (i.e. $R <: S$), the self-variable x is treated as a free variable which has an upper bound given by R. A variable is equivalent to itself, so two expressions involving x are equivalent if x occurs in the same place in both expressions. For example:

```
μx {                                μx {
 a:: Int;                            a:: Object;
 b:: λ(y: x.a). x.a;        <:       b:: λ(y: x.a). x.a;
}                                   }
```

This result may seem somewhat surprising. In the first record, b represents the arrow type $\mathsf{Int} \to \mathsf{Int}$, whereas in the second record, it is $\mathsf{Object} \to \mathsf{Object}$. The first definition of b is not a subtype of the second when projected from its enclosing record. Because of this, the rules for subtyping state that if R and S are records, then $R.l <: S.l$ only if $R \equiv S$. The more usual assumption – that $R.l <: S.l$ if $R <: S$, is not type-safe if self-references occur in covariant positions.

4. Additional Examples

This section contains some larger examples which illustrate how the DEEP calculus can be used to emulate standard OO constructs, such as classes, methods, and generics. Along the way, we will point out ways in which the DEEP calculus resolves some tricky typing issues.

For these larger examples, we use the following syntax sugar. The syntax

```
method1(x: argType, ...) = body;
method2(y: argType, ...): resultType = body;
```

is short-hand for:

```
method1 = λ(x: argType, ...). body;

method2: λ(y: argType, ...). resultType =
         λ(y: argType, ...). body;
```

The examples in this section also follow the convention that the variable g stands for the enclosing global scope.

4.1 Classes and inheritance

Classes and methods are defined much as one might expect:

```
μg{
  Point :: μthis{
    x: Int = _;
    y: Int = _;
```

```
    rSquare: Int =
      this.x * this.x +
      this.y * this.y;

    equals(that: ↑this): Bool =
      this.x == that.x  &&
      this.y == that.y;
  };

  // inheritance
  Point3 :: g.Point &* μthis{
    z: Int = _;

    rSquare: Int =
      this.x * this.x +
      this.y * this.y +
      this.z * this.z;

    equals(that: ↑this): Bool =
      this.x == that.x  &&
      this.y == that.y  &&
      this.z == that.z;
  };

  // instantiation -- in Java this would be
  // Point origin = new Point3(0,0,0);
  origin: ↑g.Point = g.Point3 &* μthis{
    x: Int = 0;
    y: Int = 0;
    z: Int = 0;
  };

  p1: ↑g.Point = g.Point3 &* μthis{
    x: Int = 1;
    y: Int = 1;
    z: Int = 1;
  };

  // using origin as a prototypical object
  p2 = g.origin &* μthis{ y: Int = 1; };

  // self-types
  cp_err = origin.equals(p1);  // type error!
  cp_ok  = origin.equals(p2);  // ok
}
```

This example defines a simple Point class. It has three data members: x, y, z, and two methods: rSquare and equals. The members of a class are immutable, so there is no difference between a data member and a method that takes no arguments.

Point3 is a class which inherits from Point. It adds a new field named z, and it overrides rSquare and equals with new definitions.

There are two ways to create a point object. First, it is possible to instantiate a class directly by using composition to assign values to data members. This is how origin and p1 are defined. Second, a new point can also be created by using an existing value as a prototypical object, and then overriding data members (or methods!) with new definitions. In the example above, p2 uses origin as a prototypical object.

Notice that inheritance and instantiation are both performed with the same operation: &*. This mechanism highlights the prototype model, in which instances are subtypes of classes.

4.1.1 Interface Types

Unlike typical OO languages, Point3 is *not* a subtype of Point, because it overrides the definitions of rSquare and equals. Subtyping between records includes the complete definition of all slots. In order for one record to be a subtype of another, it must have the exact same implementation for all methods.

Instead, Point3 and origin are subtypes of ↑Point — the *interface type* of Point. This relationship results from the identity introduced earlier:

$$t \mathbin{\&*} u \;<:\; \uparrow t \,\&\, \uparrow u$$

The distinction between class objects and interfaces highlights the dual role that classes play in OO programs. Point is a generator for both instances and subclasses, while ↑Point is the bounding type for objects that it generates.

4.1.2 Self Types

Another point of interest (pun intended) is the use of ↑this as a bounding type for the argument of equals. The expression ↑this is a *self-type*, which means "the type of this". It is an upper bound for all records which have the same interface as this. The expression ↑this emulates the "MyType" construct proposed by Bruce [13].

The expression cp_err is not well-typed, because origin and p1 have an upper bound of ↑Point. There is thus no guarantee that they have the same interface. They both happen to be 3D points in this example, but one could easily be overridden with a 2D point instead.

However, it *is* safe to compare origin and p2, because the type of p2 is ↑origin rather than ↑Point, and it is thus guaranteed to have the same interface.

4.2 Generics

A generic class is a function that takes a type as an input, and produces a class as an output. Since functions in DEEP operate over prototypes, generics can be implemented as ordinary functions.

```
μg{
  List(T: Object) = μthis {
    isEmpty :: Bool;
    head: T         = _;
    tail: g.List(T) = _;
  };

  nil(T: Object) =
    g.List(T) & μthis{
      isEmpty         = true;
      tail: g.List(T) = g.nil(T);
    };

  cons(T: Object, hd: T, tl: g.List(T)) =
    g.List(T) & μthis{
      isEmpty         = false;
      head: T         = hd;
      tail: g.List(T) = tl;
    };
}
```

This is a fairly standard implementation of parametric polymorphism, in the informal sense of the word. The List class is parameterized by a type T. Every function which operates on lists (i.e. nil and cons) must also be parameterized by T. Some form of pattern matching which infers the appropriate T would obviously be desirable in a practical implementation, but we do not consider pattern matching in the core calculus.

The only thing about these definitions which is peculiar to DEEP is the fact the type parameter T is passed as an ordinary function argument. Traditional type systems distinguish between functions over types, and functions over objects. The former are evaluated at compile-time, while the latter are evaluated at run-time.

The DEEP calculus only has one kind of function: functions over prototypes. In order to type-check the above code, the compiler will *partially evaluate* expressions of the form List(T) at compile-time. Partial evaluation is an integral part of the type system.

4.3 Virtual Types

Virtual types have been proposed as an alternative to parametric polymorphism [48]. The following example shows how an abstract array class can be defined and used. This example plugs the infamous type safety problem with Java arrays.

The following definition of arrays differs from the earlier list example because arrays have a *covariant* type parameter. An array of Int is a subtype of an array of Object. The same is not true of lists: List(t) <: List(t') only if $t \equiv t'$.

```
μg{
  Array = μthis {
    length   :: Int;
    ElemType :: Object;

    get(i: Int): this.ElemType = _;
    set(i: Int, e: this.ElemType): ↑this = _;
  }

  copyElem(a: g.Array, n: Int, m: Int): ↑a =
    a.set(m, a.get(n));

  setElem(a: g.Array & μx{ ElemType = Float; },
          n: Int, f: Float): ↑a =
    a.set(n, f);
}
```

An array holds a certain number of elements, all of which have an upper bound given by ElemType. The methods get(i) and set(i,e) will get and set the i^{th} element of the array. Since DEEP does not have a mutable heap, set returns a new array which has the same interface as the original. (Additional trickery, such as monads [52] or uniqueness types [5], can be used to avoid allocating a new array.)

The copyElem function will copy the value index n to index m. This routine is well-typed because get and set are both acting on the same array. The expression a.set requires an argument of type a.ElemType — a dependent type. The expression a.get(m) has type a.ElemType, so the types match up.

There is practical problem with using covariant array types. Although it is safe to copy elements within the same array, it is not safe to put elements from any other source into the array. The definition of setElem circumvents this problem by giving ElemType a *final bound* of Float. Since a.ElemType is equivalent to Float, it is safe to put any floating-point number into the array.

4.4 The Expression Problem:

Our final example presents a solution to the expression problem in DEEP. This solution closely mimics the pseudo-code given in the introduction, but includes a bit more detail. We demonstrate that it is possible to add a new class, add a new operation, and to mix the two extensions together.

```
μg{
  // core definitions
  EBase = μm {
    // empty class declaration
    Expr :: μthis{ };

    Literal :: m.Expr &* μthis{
      value: Int = _;
    };

    Plus :: m.Expr &* μthis{
      e1: ↑m.Expr = _;
      e2: ↑m.Expr = _;
    };
  };

  // add a new operation
  EvalMod = g.EBase &* μm{
    Expr :: μthis{
      eval: Int = _;
    };

    Literal :: μthis{
      eval: Int = this.value;
    };

    Plus :: μthis{
      eval: Int = this.e1.eval + this.e2.eval;
    };
  };

  // add a new type
  MultMod = g.EBase &* μm{
    Times :: m.Expr &* μthis{
      e1: ↑m.Expr = _;
      e2: ↑m.Expr = _;
    };
  };

  // combine the two extensions
  EvalMult = g.MultMod &* g.EvalMod &* μm{
    Times :: μthis{
      eval: Int = this.e1.eval * this.e2.eval;
    };
  };
};
```

EBase is a module with three nested classes — Expr, Literal, and Plus, that together define a simple syntax tree. Literal and Plus inherit from Expr.

EvalMod is a module which extends EBase and adds a new method — eval — to all three classes. MultMod also extends EBase, and adds a new class — Times — which inherits from Expr.

The EvalMult module combines these two extensions by composing EvalMod with MultMod. However, the pure composition (EvalMod &* MultMod) has a gap in functionality because the Times class does not implement the eval method. The definition of EvalMult fills in this gap.

4.5 Limitation – no indexed data types

DEEP does have a major limitation. One of the main applications of dependent types to date has been reasoning about the sizes of data structures. Instead of ordinary lists, dependent type system can encode lists of length n, where n is natural number. Using this type, is is possible to prove that operations such as map and reverse preserve the length of lists, that concatenating two lists will add their lengths, etc.

In contrast with other dependent type systems, such as DML [54] and Epigram [40], DEEP does not provide good support for types of this nature. It is certainly possible to extend the definition of List given earlier with an integer parameter. However, in order to reason about sizes, the type system must have knowledge of basic arithmetic identities, such as the commutativity and associativity of addition. List(n + m) should be the same type as List(m + n).

DEEP does not handle such identities. This limitation means that although the core of DEEP supports dependent types, in practice it is not able take full advantage of it.

5. Subtyping

Subtyping is the least relation between terms which is closed under the definitions in figure 2. Figure 2 defines three relations over terms; where "$\lhd$" is a meta-variable that ranges over all three.

- $t <: u$ means that t is a subtype of u.

Subtyping (congruence): $\boxed{\Gamma \vdash t \lhd t}$

$$\frac{\Gamma \vdash t \lhd u \quad u \lhd s \quad u \text{ wf}}{\Gamma \vdash t \lhd s} \qquad \frac{\Gamma \vdash u \equiv t}{\Gamma \vdash t \equiv u} \quad \text{(S-TRANS), (S-SYMM)}$$

$$\frac{\Gamma \vdash t \equiv u}{\Gamma \vdash t \lessdot u} \qquad \frac{\Gamma \vdash t \lessdot u}{\Gamma \vdash t <: u} \quad \text{(S-WEAKEN1-2)}$$

$$\Gamma \vdash x \equiv x \qquad \frac{\Gamma \vdash t \lhd t', \quad u \equiv u'}{\Gamma \vdash t(u) \lhd t'(u')} \quad \text{(S-VAR), (S-APPLY)}$$

$$\frac{\Gamma \vdash t \equiv t'}{\Gamma \vdash t.l \equiv t'.l} \qquad \Gamma \vdash t \lessdot \uparrow t \quad \text{(S-PROJECT), (S-IFACEINTRO)}$$

$$\frac{\Gamma \vdash t \lessdot u}{\Gamma \vdash \uparrow t \equiv \uparrow u} \qquad \frac{\Gamma \vdash t <: u}{\Gamma \vdash \uparrow t <: \uparrow u} \quad \text{(S-INTERFACE1-2)}$$

$$\frac{\Gamma \vdash t \equiv t' \quad u \equiv u'}{\Gamma \vdash t \otimes u \equiv t' \otimes u'} \quad \text{(S-COMPOSE)}$$

$$\frac{\Gamma \vdash t \equiv t' \qquad \Gamma, x <: t \vdash u \lhd s}{\Gamma \vdash \lambda(x : t).\, u \lhd \lambda(x : t').\, s} \quad \text{(S-FUNCTION)}$$

$$\frac{\text{dom}(\overline{d})\ [\lhd]\ \text{dom}(\overline{e}) \qquad \forall \ell \in \text{dom}(\overline{e}).\ \Gamma, x <: \mu x\{\uparrow \overline{d}\} \vdash d_\ell \lhd e_\ell}{\Gamma \vdash \mu x\{\overline{d}\} \lhd \mu x\{\overline{e}\}} \quad \text{(S-RECORD)}$$

$$[<:] \stackrel{\text{def}}{=} \supseteq \qquad [\lessdot] \stackrel{\text{def}}{=} = \qquad [\equiv] \stackrel{\text{def}}{=} =$$

Declaration subtyping: $\boxed{\Gamma \vdash d \lhd d}$

$$\frac{\Gamma \vdash d \equiv e}{\Gamma \vdash d \lessdot e} \qquad \frac{\Gamma \vdash d \lessdot e}{\Gamma \vdash d <: e} \quad \text{(DS-WEAKEN1-2)}$$

(DS-DECL1-6)

$$\frac{\Gamma \vdash t \lhd u}{\Gamma \vdash (l :: t) \lhd (l :: u)} \qquad \frac{\Gamma \vdash t \equiv t'}{\Gamma \vdash (l : t = \_) \equiv (l : t' = \_)}$$

$$\frac{\Gamma \vdash t \equiv u}{\Gamma \vdash (l = t) \equiv (l = u)} \qquad \frac{\Gamma \vdash t \equiv t', \quad u \equiv u'}{\Gamma \vdash (l : t = u) \equiv (l : t' = u')}$$

$$\frac{\Gamma \vdash t <: u}{\Gamma \vdash (l = t) <: (l :: u)} \qquad \frac{\Gamma \vdash t \equiv t'}{\Gamma \vdash (l : t = u) \lessdot (l : t' = \_)}$$

Subtyping (reduction): $\boxed{\Gamma \vdash t \lhd t}$

$$\frac{x \lhd t \in \Gamma}{\Gamma \vdash x \lhd t} \quad \text{(SE-VAR)}$$

$$\frac{\Gamma \vdash t \lhd \lambda(x : u').\, s}{\Gamma \vdash t(u) \lhd [x \mapsto u]\, s} \quad \text{(SE-APPLY)}$$

$$\frac{t \lhd' \mu x\{\overline{d}\} \qquad \text{declType}(\lhd', d_\ell, \lhd, u)}{\Gamma \vdash t.\ell \lhd [x \mapsto t]\, u} \quad \text{(SE-PROJECT)}$$

$$\begin{array}{llll} \text{declType}(\lhd, & l :: t, & \lhd, & t) \\ \text{declType}(\lhd, & l = t, & \equiv, & t) \\ \text{declType}(\lhd, & l : t = \_, & <:, & t) \\ \text{declType}(\lhd, & l : t = u, & <:, & t) \\ \text{declType}(\lhd, & l : t = u, & \equiv, & u) \end{array}$$

$$\frac{\Gamma \vdash t \equiv t'}{\Gamma \vdash \lambda(x : t).\, u \otimes \lambda(x : t').\, s \equiv \lambda(x : t').\, (u \otimes s)} \quad \text{(SE-COMPOSEFUN)}$$

$$\frac{\Gamma, x <: \mu x\{\uparrow \overline{e}\} \vdash \overline{c} \otimes \overline{d} \Rrightarrow \overline{e}}{\Gamma \vdash \mu x\{\overline{c}\} \otimes \mu x\{\overline{d}\} \equiv \mu x\{\overline{e}\}} \quad \text{(SE-COMPOSEREC)}$$

$$\frac{\uparrow v \longrightarrow w}{\Gamma \vdash \uparrow v \equiv w} \quad \text{(SE-INTERFACE)}$$

$$\begin{array}{lll} \Gamma \vdash & (t \,\&\, u) \equiv (u \,\&\, t) & \text{(SE-ISECTCOMM),} \\ \Gamma \vdash & (t \,\&\, u) \,\&\, s \equiv t \,\&\, (u \,\&\, s) & \text{(SE-ISECTASSOC)} \end{array}$$

$$\frac{\Gamma \vdash t <: u}{\Gamma \vdash t \,\&\, u \equiv t} \qquad t \,\&\, u <: t \quad \text{(SE-ISECTELIM1-2)}$$

$$\Gamma \vdash \uparrow(t \mathbin{\&*} u) \equiv \uparrow t \,\&\, \uparrow u \quad \text{(SE-IFCOMPOSE)}$$

Checked composition: $\boxed{\Gamma \vdash c \otimes d \Rrightarrow e}$

$$\Gamma \vdash (l :: t) \otimes (l :: u) \Rrightarrow (l :: t \otimes u)$$

$$\frac{\Gamma \vdash e <: d \qquad e \text{ replaces } d}{\Gamma \vdash d \,\&\, e \Rrightarrow e} \qquad \frac{\Gamma \vdash d <: e \qquad d \text{ overrides } e}{\Gamma \vdash d \,\&\, e \Rrightarrow d}$$

$$\frac{\Gamma \vdash e <: \uparrow d \qquad e \text{ replaces } d}{\Gamma \vdash d \mathbin{\&*} e \Rrightarrow e} \qquad \frac{\Gamma \vdash d <: \uparrow e \qquad d \text{ overrides } e}{\Gamma \vdash d \mathbin{\&*} e \Rrightarrow d}$$

Figure 2. Subtyping. (Note that $\lhd$ is a metavariable which ranges over $<:$, $\lessdot$, and $\equiv$.)

- $t \equiv u$ means that t is type equivalent to u.
  $t \equiv u$ if and only if $t <: u$ and $u <: t$.

- $t \lessdot u$ means that t is an *exact* subtype of u.
  $t \lessdot u$ only if $t <: u$ and $\uparrow t \equiv \uparrow u$.

Exact subtypes are used to prove that a class which inherits from a virtual class is well-formed. It is discussed in more detail in section 6.1.

5.1 Subtyping as Partial Evaluation

Since terms in DEEP range over both types and objects, the "$\equiv$" relation not only defines equivalence between types, but also equality between objects. Moreover, equivalence is defined over arbitrary expressions, not just values. Two expressions are equivalent if they can be reduced to a common term.

The subtyping rules are split into two groups. The *congruence rules*, labeled S-*, compare terms which have the same shape. The *reduction rules*, labeled SE-*, mirror the definition of untyped reduction in the operational semantics. As we will discuss shortly, the reduction rules mean that subtyping can be seen as a form of partial evaluation.

Subtyping differs from untyped reduction in three main ways. First, composition is restricted in order to eliminate invalid compositions. The restrictions are:

- Functions cannot be composed unless they have equivalent argument types.
- Records cannot be composed unless their slots have compatible types. For example, the composition of $\mu x\{l = 3;\}$ and $\mu x\{l = 4;\}$ is illegal, because $3 \not\equiv 4$.

Second, subtyping includes some additional algebraic identities. Type intersection is commutative, associative, and obeys the absorption law for semi-lattices.

Third, subtyping can discard information. The judgment $t \equiv u$ is similar to ordinary evaluation. It means that t can be reduced to an equivalent term u (or vice versa). The judgment $t <: u$ is similar to typing. It means that t can be *simplified* to u, where u is an upper bound for t.

Free variables can be simplified to their upper bounds using rule SE-VAR. Any expressions that involve free variables can also be simplified to an upper bound by applying the reduction rules. Although we refer to this mechanism as "partial evaluation", it can also be regarded as a primitive form of *abstract interpretation*.

To see how this works, consider the rule SE-PROJECT:

- If $t <: \mu x\{ l :: Int;\}$, then $t.l <: Int$.
- If $t \equiv \mu x\{ l :: Int;\}$, then $t.l \equiv Int$.
- If $t <: \mu x\{ l = Int;\}$, then $t.l \equiv Int$.
- If $t <: \mu x\{ l : Int = 3;\}$ then $t.l <: Int$, and $t.l \equiv 3$.

This definition serves as both a reduction rule, and a typing rule. If we know the type of t, then we can find the type of $t.l$. If we know the exact value of t, then we can find the exact value of $t.l$. Most interestingly, we can also find the exact value of $t.l$ if l is final bound, even if the only thing we know about t is its type.

5.2 True Partial Evaluation

This ability to derive exact values as well as upper bounds is the reason why we refer to this mechanism as "partial evaluation" rather than "typing". Consider the following two definitions of a power function, which computes x^n.

```
μg{
  powerT(n: Int, x: Int): Int =
    if (n == 0) then 1
    else x * g.powerT(n−1,x);

  powerE(n: Int, x: Int) =
    if (n == 0) then 1
    else x * g.powerE(n−1,x);

  pt = g.powerT(3,x); // has type Int
  pe = g.powerE(3,x); // has type x*x*x*1
}
```

Recall that the definition of powerT is syntax sugar for:

```
powerT: λ(n: Int). λ(x: Int). Int = ...
```

This definition is a field, which means that powerT has an explicit bounding type: a function which ignores its arguments and returns Int. When the compiler type-checks the expression powerT(3,x), it will apply this bounding type using rule SE-APPLY. The result of the application is Int, which is the type of the expression. In this case the process is very similar to what happens in an ordinary type system.

The definition of powerE, on the other hand, is a final binding rather than a field, so it has no bounding type other than itself. The compiler has no choice but to expand the body of powerE, and simplify the result to a value. In the process of doing so, it will reduce the if expressions, and expand further recursive calls. Assuming that x is a free variable of type Int, the final derivation is:

```
g.powerE(3,x) ≡ x*x*x*1 <: Int
```

This derivation yields the same upper bound as the one for powerT, but it has partially evaluated the power function in the process. Since g.powerE(3,x) is provably equivalent to x*x*x*1, the latter can be substituted for the former in compiled code.

Limitations The price that must be paid for integrating a partial evaluator into the type system is loss of decidability. If the type checker is allowed to expand recursive function calls, then there is no guarantee that type checking will terminate. In section 6.9, we sketch an algorithm that ensures termination by preventing the compiler from entering recursive calls. However, this restriction would make the definition of powerE illegal.

As with dependent types, DEEP possesses the machinery to perform partial evaluation, but is currently unable to take full advantage of it.

5.3 Nominal Subtyping

The subtype relation is defined over all terms, not just normal forms. One benefit of defining subtypes in this way is that the *nominal* typings used in most OO languages can be seen as a special case of *structural* comparisons between terms.

Featherweight Java [32] is a good example of how nominal subtyping is typically implemented in mainstream OO languages. A program in featherweight Java consists of a pair (CT, e), where CT is a global class table, and e is an expression. Types in featherweight Java are the names of classes in the table. Subclass relationships are read directly from the table, which is fixed.

The DEEP calculus does not use a global class table. Instead, subtype relationships are inferred from the local typing context. To see how nominal subtyping works in practice, consider the following code, which is an excerpt from the Point example given earlier:

```
μg{
  Point :: μthis{ ... };

  // inheritance
  Point3 :: g.Point &* μthis{ ... };
}
```

The name of a class is a *path* of the form $x.l_1.l_2...l_n$. In the code above, g.Point and g.Point3 are class names. Both of them are declared as virtual classes, so their exact definitions are not statically known. However, it is possible to infer a subtype relation between the two by comparing paths. We prove that ↑g.Point3 <: ↑g.Point as follows:

↑g.Point3	<:	↑(g.Point &* {...})	SE-PROJECT, SE-VAR
	≡	↑g.Point & ↑{...}	SE-IFCOMPOSE
	<:	↑g.Point	S-ISECTELIM2

This derivation expands g.Point3 to its definition by finding the type of g in the local context. The remaining steps exploit the relationship between interfaces and compositions. The derivation is nominal because it only compares paths; at no point is there any need to compare records.

5.4 Match Subtyping

Subtyping between records in DEEP is slightly different from that traditionally used in the literature [19]. A record $\mu x\{\overline{d}\}$ is a recursive structure. As such, every record has an infinite expansion denoted by the fixpoint of the record. In a traditional type system,

subtyping between recursive structures is coinductively defined as a comparison between infinite expansions.

In DEEP, however, a comparison between $\mu x\{\overline{d}\}$ and $\mu x\{\overline{e}\}$, does not compare the infinite expansions of the two records. Instead, it compares their definitions, where x is taken to be a free variable with an upper bound of $\mu x\{\uparrow\overline{d}\}$. If two records are in a subtype relation, then they have the *same recursive structure* [19]. Bruce calls this mechanism *match subtyping* [13].

Match subtyping is not ordinarily a sound basis for a static type system unless that type system supports some form of dependent types. In a non-dependent type system, the type of a term can depend only on the static types of its subterms. This restriction leads to the following typing rule for projecting slots from records. (The syntax here assumes that T and U range over types, X is a type variable, and $\overline{D}$ is a sequence of declarations of the form $l : U$).

$$\frac{\Gamma \vdash t : T, \quad T = \mu X\{\overline{D};\, l : U;\}}{\Gamma \vdash t.l : [X \mapsto T]\, U} \qquad \text{(non-dependent typing)}$$

This definition unfolds the recursive type T by replacing all occurrences of the self-type X with T. The type T is the *static type* of the term t. The static type is only an upper bound for the *exact type* [13] of t, which is not necessarily known at compile-time. With this rule, the use of match subtyping would lead to a type error if X occurs in a contravariant or invariant position, such as the argument type of a method.

To see why this creates a problem, consider the following Java-like pseudo-code.

```
class Point μX {
  ...
  Bool equals(X that) { ... }
}

class Point3 extends Point μX { ... }

Point origin = new Point3(0,0,0);  // error
Point p1     = new Point(1,1);

... origin.equals(p1); ...    // unsafe call
```

In this example, the origin variable has a static type of Point. The static type of origin.equals is a function from Point → Bool, and the compiler would therefor judge that origin.equals(p1) is well-typed. At run-time, however, origin has an actual type of Point3, which cannot be compared with a 2D point.

The only way to avoid this situation in a traditional type system is to ensure that Point3 is not a subtype of Point, which would make the definition of origin ill-typed.

5.5 Dependent Types

Compare the non-dependent type rule above with the dependently-typed definition used in DEEP:

$$\frac{\Gamma \vdash t <: \mu x\{\overline{d};\, l :: u;\}}{\Gamma \vdash t.l <: [x \mapsto t]u} \qquad \text{SE-PROJECT}$$

This definition replaces all occurrences of the self-variable x with the term t. This replacement is safe for use with match subtyping, because it does not discard any information about the exact run-time type of t. Referring back to the earlier Point example, we have:

```
μg{
  Point :: μthis{
    equals(that: ↑this): Bool = ...;
  };

  Point3 :: g.Point &* μthis{ ... };
  origin: ↑g.Point = ...
  p1:     ↑g.Point = ...

  ... g.origin.equals(p1) ... // type error
}
```

In this example, the static type of g.origin.equals is a function: λ(that: ↑g.origin). Bool. Passing p1 as an argument to this call will result in a type error unless p1 is a subtype of ↑g.origin, which it is not. As a side benefit, the type error now occurs at the site of the unsafe call, rather than at the declaration of origin.

In addition to dependent record types, DEEP supports dependent function types. A dependent function type is one in which the value of the argument appears in the type of the result. The Array example shown earlier illustrates the use of such functions.

6. Type Safety

In an ordinary type system, a term is considered well-typed if a type can be assigned to it. In DEEP every term is already a type, so the notion of being well-typed does not make much sense. Instead, terms are judged to be well-formed. Terms which are well-formed cannot "go wrong"; that is, they cannot generate type errors at run-time. A type error is one of the following events:

- A program attempts to call a function with a value of the wrong type.
- A program attempts to project a slot that does not exist.
- A program attempts to compose two functions with different argument types.
- A program attempts to compose two records, but the slots have incompatible definitions.

Ensuring that function calls are well-typed, and that slots exist is straightforward. However, verifying composition turns out to be much more difficult. Loosely speaking, a record is considered to be well-formed if the logical constraints established by its declarations are satisfiable. Given this definition, composition has a most unfortunate property:

The composition of two well-formed terms is not necessarily well-formed.

There are several ways in which a composition may be ill-formed. Two records may define different final bindings for the same slot, or they may define different ranges for the same field. A record is also ill-formed if there are cycles of dependencies between its slots that prevent the proof of well-formedness from terminating, and such cycles can be introduced by composition. All these cases are illustrated in the following example:

```
μx{ a = 3;              μx{ a = 4;
    b: Bool = _;            b: Int = _;
    c :: μy{ };     ⊗       c :: x.d;
    d :: x.c;               d :: μz{ };
  }                       }
```

6.1 Well-formedness

The rules for well-formedness are given in figure 3. A function application $t(u)$ is well formed if t is a well-formed function, and u is a well-formed term of the appropriate type. A record projection $t.l$ is well-formed if t is a well-formed record and the slot l exists.

The rules for composition and intersection are more subtle. A composition $t \mathbin{\&*} u$ is well formed if $\uparrow t \mathbin{\&} \uparrow u$ is well-formed. Intuitively, this follows from the fact that the $\&*$ operator will *override* field implementations. It is not possible for there to be a type clash between the implementations of two fields, because one simply overrides the other. All type clashes must occur in the

Well-formed terms: $\boxed{t \text{ wf}}$

$$\frac{x \lhd t \in \Gamma}{\Gamma \vdash\ x \text{ wf}} \quad \text{(W-VAR)}$$

$$\frac{\Gamma \vdash\ t \text{ wf}, \quad u \text{ wf} \quad \Gamma \vdash\ t <: \lambda(x : u').\ s, \quad u <: u'}{\Gamma \vdash\ t(u) \text{ wf}} \quad \text{(W-APPLY)}$$

$$\frac{\Gamma \vdash\ t \text{ wf}, \quad t <: \mu x\{\overline{d}\}, \quad l \in \text{dom}(\overline{d})}{\Gamma \vdash\ t.l \text{ wf}} \quad \text{(W-PROJECT)}$$

$$\frac{\Gamma \vdash\ t \text{ wf}, \quad u \text{ wf} \quad \Gamma \vdash\ t \,\&\, u \equiv s, \quad s \text{ wf}}{\Gamma \vdash\ t \,\&\, u \text{ wf}} \quad \text{(W-ISECT)}$$

$$\frac{\Gamma \vdash t \text{ wf}}{\Gamma \vdash \uparrow t \text{ wf}} \quad \frac{\Gamma \vdash\ \uparrow t \,\&\, \uparrow u \text{ wf}}{\Gamma \vdash\ t \mathbin{\&*} u \text{ wf}} \quad \text{(W-INTERFACE) (W-COMPOSE)}$$

$$\frac{\Gamma \vdash\ t \text{ wf}, \quad \uparrow t \mathbin{\&*} v \equiv s, \quad s \text{ wf}}{\Gamma \vdash\ t \mathbin{\&*} v \text{ wf}} \quad \text{(W-INHERIT)}$$

$$\frac{\Gamma \vdash\ t \text{ wf} \quad \Gamma, x <: t \vdash\ u \text{ wf}}{\Gamma \vdash\ \lambda(x : t).\ u \text{ wf}} \quad \text{(W-FUNCTION)}$$

$$\frac{\forall \ell \in \text{dom}(\overline{d}).\ \Gamma, x \lessdot \mu x\{\uparrow \overline{d}\} \vdash\ d_\ell \text{ wf}}{\Gamma \vdash\ \mu x\{\overline{d}\} \text{ wf}} \quad \text{(W-RECORD)}$$

Well-formed declarations: $\boxed{d \text{ wf}}$

$$\frac{\Gamma \vdash\ t \text{ wf}}{\Gamma \vdash\ (l :: t) \text{ wf}} \quad \frac{\Gamma \vdash\ t \text{ wf}}{\Gamma \vdash\ (l = t) \text{ wf}} \quad \text{(W-DECL1-2)}$$

$$\frac{\text{wk}(\Gamma) \vdash\ t \text{ wf}}{\Gamma \vdash\ (l : t = \_) \text{ wf}} \quad \text{(W-FIELD1)}$$

$$\frac{\text{wk}(\Gamma) \vdash\ t \text{ wf}, \quad u \text{ wf}, \quad u <: t}{\Gamma \vdash\ (l : t = u) \text{ wf}} \quad \text{(W-FIELD2)}$$

$$\begin{array}{lcl} \text{wk}(\emptyset) & = & \emptyset \\ \text{wk}(\Gamma, x \lhd t) & = & \text{wk}(\Gamma), x <: t \end{array}$$

Figure 3. Well-formedness

interface, so if the interface of the composition is well-formed, then the composition as a whole is well-formed.

The rule for intersection states that a term $t \,\&\, u$ is only well-formed if it is equivalent to some other well-formed term. That "other term" cannot be an intersection, or the proof will fail to terminate. Referring back to the rules for subtyping, there are only two ways in which an intersection can be eliminated:

$$\frac{\Gamma \vdash\ t <: u}{\Gamma \vdash\ t \,\&\, u \equiv t} \quad \text{(Absorption)}$$

$$\frac{\Gamma \vdash\ t \equiv v_t, \quad u \equiv v_u, \quad v_t \,\&\, v_u \equiv w}{\Gamma \vdash\ t \,\&\, u \equiv w} \quad \text{(Reduction)}$$

6.2 Instantiation

Absorption handles the case of pure specialization, where one term is a direct subtype of the other. When used with "&*", it also handles class instantiation. A class is instantiated by overriding existing fields with new implementations. Consider the following composition:

```
g.Point3 &* {
  x: Int = 0; y: Int = 0; z: Int = 0;
};
```

In this case, x, y, and z are already defined within g.Point3, so it is possible to prove that:

$$\uparrow \texttt{g.Point3} <: \uparrow\{\ \texttt{x: Int} = 0;\ \texttt{y: Int} = 0;\ \texttt{z: Int} = 0;\ \}$$

and the composition is therefore well-formed. In other words, instantiating a virtual class is an operation which can be safely performed at run-time.

6.3 Inheritance

Reduction is used to handle inheritance. A composition is $t \otimes u$ is well-formed if it can be partially-evaluated to a well-formed record. Since both partial evaluation and the proof of well-formedness are done at compile-time, this means that inheritance is a compile-time operation. DEEP is no different from other statically-typed OO languages in this regard. DEEP also provides a faithful model of feature composition, which is likewise performed by generating code at compile-time.

The rule W-INHERIT handles the common syntax for inheritance which is used throughout the examples in this paper. The right-hand side of the composition is a literal record which refines some superclass. The declarations in this record can only be understood within the context of the superclass, so the composition should be performed *before* the record is checked for well-formedness. This rule could be removed without changing the fundamental expressiveness of the system, but it makes code less verbose by eliminating redundant declarations.

6.4 Exact Types

Unfortunately, the use of partial evaluation places a strong limit on inheritance. In order to verify that $t \mathbin{\&*} u$ is well-formed, both $\uparrow t$ and $\uparrow u$ must be known up to equivalence. Without some additional extension, it would only be possible to inherit from a class which is statically known — i.e. one which is final bound. That would rule out the possibility of inheriting from a virtual class.

In order to overcome this problem, DEEP introduces an *exact type* relation between terms, which is written "$\lessdot$". The judgment $t \lessdot u$ implies that $t <: u$ and $\uparrow t \equiv \uparrow u$. If a term has an exact type which is statically known, then the interface of that term is known up to equivalence, and it is possible to prove that compositions involving that term are well-formed.

Furthermore, rule W-RECORD states that a record $\mu x\{\overline{d}\}$ is well-formed if its declarations are well-formed under the assumption that $x \lessdot \mu x\{\uparrow \overline{d}\}$. In other words, when verifying a class C, instead of assuming that this is a subtype of C, we assume that this has exact type C. Since the enclosing record has an exact type, any nested virtual classes also have exact types. It is safe to inherit from a virtual class.

Verifying a class C under the assumption that $\texttt{this} \lessdot C$ ensures that all instances of C are valid. It is thus safe to create instances of C at run-time. Unfortunately, subclasses of C may invalidate certain declarations. Consider the following example:

```
μg{
  Mod1 = μthis{
    A :: { a :: Int; };
    B :: this.A &* { a = 4; };
  };
```

```
  Mod2 = g.Mod1 &* μthis{
    A :: { a = 3; };
    // error -- B is now invalid
  };
}
```

In this example, Mod1 contains two classes A and B, where B inherits from A. Inheriting from a virtual class is a potentially dangerous operation, because the interface of A is not fixed. Nevertheless, exact types allow us to prove that Mod1 is well-formed.

However, when Mod2 extends Mod1 we have a problem. Mod2 changes the definition of $A.a$ from a virtual binding to a final binding. The definition of A is still okay, but the inherited copy of B is now invalid.

This error will be detected at compile-time because inheritance is a compile-time operation. The only way to verify Mod2 is to generate the complete record, and check all declarations, *including declarations which are inherited from Mod1.* We check the definition of B when it is first declared within Mod1, and then re-check the interface of B again within Mod2.

6.5 Separate Compilation

At first glance, this might seem like a bad idea, since appears to rule out the possibility of separate compilation. However, the type system only needs to re-check the *interface* of inherited declarations.

Field implementations are checked once — at the point where they are initially declared. Consequently, well-formedness for fields is performed within a weaker context in which $\mathsf{this} <: C$, rather than $\mathsf{this} \lessdot C$. Using a weaker context ensures that the implementation of fields can be safely inherited by subclasses without re-checking them.

6.6 Summary

It is safe to inherit from a virtual class. However, it is not safe to inherit from a class which is stored in a field, or passed as the argument to a function, because the exact type of such a class is unknown. It is always safe to instantiate a virtual class, regardless of whether it has an exact type.

The use of exact types means that every subclass has to re-check all of the declarations that it inherits from super-classes. However, it does *not* need to re-check any implementations that it inherits. DEEP thus supports separate compilation of modules.

6.7 Transitivity Elimination

Before moving on to the formal proof of type safety, there is one more technical issue regarding subtyping that needs to be addressed.

Figure 2 defines the subtype relation in a declarative manner. This presentation is more concise, but it does not specify an algorithm for determining whether one term is a subtype of another. As it turns out, the subtype relation must be split into two different algorithms, because a proof of well-formedness uses the relation in two distinct ways.

First, there is the obvious use: subtyping compares terms. Given two terms t and u, we wish to know whether $t \lhd u$. ("$\lhd$" is a meta-variable that ranges over $<:$, $\lessdot$, and $\equiv$.)

Second, subtyping is used as a replacement for typing. Given a single term t, we wish to find the "type" of t — a minimal upper bound v such that $t <: v$. This judgment is used in rules W-APPLY and W-PROJECT.

Due to lack of space, we do not present a full definition of both algorithms here. The full definition is available as an appendix in the electronic version of this paper; what follows is a brief sketch.

The algorithmic definition splits subtyping into two relations: "$\lhd$" and "$\vec{\lhd}$". The judgment $\Gamma \vdash_{\text{A}} t \lhd u$ is an algorithmic comparison between two terms, and it contains the subtype congruence rules (S-*). As usual, the main challenge of defining the algorithm is eliminating the rules for symmetry and transitivity:

$$\frac{\Gamma \vdash u \equiv t}{\Gamma \vdash t \equiv u} \qquad \frac{\Gamma \vdash t \lhd u, \quad u \lhd s, \quad u \text{ wf}}{\Gamma \vdash t \lhd s} \qquad \begin{array}{l}\text{(S-SYMM),}\\ \text{(S-TRANS)}\end{array}$$

Transitivity involves "guessing" a suitable u. Even worse, it involves guessing a u which is well-formed. The proof of well-formedness involves subtyping, and that leads to a circular definition. To overcome this problem, we replace the rules for symmetry and transitivity with the following definitions:

$$\frac{\Gamma \vdash_{\text{A}} t \vec{\lhd} u, \quad u \lhd s}{\Gamma \vdash_{\text{A}} t \lhd s} \qquad \frac{\Gamma \vdash_{\text{A}} t \lhd u, \quad s \vec{\equiv} u}{\Gamma \vdash_{\text{A}} t \lhd s}$$

The judgment $\Gamma \vdash_{\text{A}} t \vec{\lhd} u$ contains the subtype reduction rules (SE-*). These rules have the advantage that they can be read from left to right, just like untyped reduction, so there is no need to "guess" u. The only reduction rule that is not included is SE-ISECTELIM2, because it discards information and introduces a non-determinism.

The judgment $\Gamma \vdash_{\text{A}} t \vec{\lhd} u$ provides a concrete algorithm for performing typing and partial evaluation. It also has the following desirable properties:

1. It is confluent.
2. If $t \vec{\lhd} \ldots \vec{\lhd} v$, then v is the minimal upper bound of t.
3. If t wf and $t \vec{\lhd} t'$, then t' wf.

The third property is especially important, since it allows us to eliminate the circular dependency between the judgments for subtyping and well-formedness. We do not have space for a formal proof here, but it is essentially the same as the proof of type safety below.

6.8 Progress and Preservation

We give a full proof of type safety using the standard techniques of progress and preservation. [53].

The canonical small-step preservation proof states that if $t : T$ and $t \longrightarrow t'$ then $t' : T$. In DEEP there is no typing judgment, so we prove instead that if t wf and $t \longrightarrow t'$, then $t \equiv t'$ and t' wf.

This definition is a faithful interpretation of the traditional meaning of type-safety. If $t \equiv t'$, then $t <: v$ implies that $t' <: v$, for any v. Thus, no matter what bounding type we assign to a program, that bounding type will be preserved under evaluation.

LEMMA 1 (Weakening).
If $\Gamma \vdash u$ wf, $u \lhd s$ and $x \notin \Gamma$, then $\Gamma, x <: t \vdash u$ wf, $u \lhd s$

Proof: straightforward induction on the derivations for subtyping and well-formedness.

LEMMA 2 (Narrowing).
If $\Gamma, x <: t \vdash u$ wf, $u \lhd s$ and $\Gamma \vdash t'$ wf, $t' <: t$
then $\Gamma, x <: t' \vdash u$ wf, $u \lhd s$

This lemma states that judgments about subtyping and well-formedness which are made in a general context still hold in a more specific context.

Proof: by induction on the derivations of subtyping and well-formedness. By lemma 1 and S-TRANS, every judgment of the form $x <: t$ can replaced with a judgment of the form $x <: t' <: t$.

LEMMA 3 (Substitution).
If $\Gamma, x <: t \vdash u$ wf, $u \lhd s$ and $\Gamma \vdash t'$ wf, $t' <: t$,
then $\Gamma \vdash [x \mapsto t']u$ wf, $[x \mapsto t']u \lhd [x \mapsto t']s$.

Proof: by induction on the derivations for subtyping and well-formedness. Every judgment of the form $x \equiv x$ can be replaced

with $t' \equiv t'$, every judgment of the form $x <: t$ can be replaced with $t' <: t$, and every judgment of the form x wf can be replaced with t' wf.

LEMMA 4 (Inversion of subtyping).
If $\Gamma \vdash v <: \mu x\{\overline{d}\}$ then there is a proof of that fact which ends in S-RECORD (i.e. v is a record).
If $\Gamma \vdash v <: \lambda(x : t).\ u$, then there is a proof that fact which ends in S-FUNCTION (i.e v is a function).

Proof: This critical result is difficult to prove in the declarative system, but follows immediately in the algorithmic system from elimination of transitivity. See section 6.7.

LEMMA 5 (Subterms are well-formed).
If $\Gamma \vdash E[u]$ wf and $u \neq v$, then $\Gamma \vdash u$ wf
Proof: by inspection of well-formedness, and induction on the structure of $E[u]$.

LEMMA 6 (Replacement).
If $\Gamma \vdash E[t]$ wf, $t \equiv u$, and u wf,
then $\Gamma \vdash E[t] \equiv E[u]$, and $E[u]$ wf.
Proof: $E[t] \equiv E[u]$ by inspection of the subtype relation, and induction on the structure of $E[t]$. $E[u]$ wf by induction on the derivation for $E[t]$ wf.

LEMMA 7 (Composition).
If $\Gamma \vdash \mu x\{\overline{c}\} <: \mu x\{\overline{d}\}$ then $\mu x\{\overline{c}\}\ \&\ \mu x\{\overline{d}\} \longrightarrow \mu x\{\overline{e}\}$ such that $\mu x\{\overline{e}\} \equiv \mu x\{\overline{c}\}$.
If $\Gamma \vdash \mu x\{\uparrow\overline{c}\} <: \mu x\{\uparrow\overline{d}\}$ then $\mu x\{\overline{c}\}\ \&*\ \mu x\{\overline{d}\} \longrightarrow \mu x\{\overline{e}\}$ such that $\mu x\{\uparrow\overline{e}\} \equiv \mu x\{\uparrow\overline{c}\}$.

Proof: This lemma holds for records if it holds between declarations. If $c_\ell\ \&\ d_\ell \longrightarrow e_\ell$ by DE-COMPOSE, then $e_\ell \equiv c_\ell$ by SE-ISECTELIM1. If c_ℓ overrides d_ℓ, then $e_\ell = c_\ell$. If d_ℓ replaces c_ℓ, then we see by inspection that $c_\ell <: d_\ell$ implies $c_\ell \equiv d_\ell$. The proof for $\&*$ is similar, since $\&*$ and $\&$ are identical with respect to interfaces.

THEOREM 1 (Progress).
If $\emptyset \vdash t$ wf then either t is a value, or $t \longrightarrow t'$ for some t'.
Proof: By induction on the derivation of t wf.

- Case $t = E[u]$, and $u \neq v$. By lemma 5, u wf. By the induction hypothesis, $u \longrightarrow u'$, and t reduces by E-CONGRUENCE.
- Case W-APPLY — $t = v(w)$. By lemma 4, v is a function, and t reduces by E-APPLY.
- Case W-PROJECT — $t = v.\ell$. By lemma 4, v is a record with a slot ℓ, and t reduces by E-PROJECT.
- Case W-ISECT — $t = v\ \&\ w$. An intersection is well-formed only if it can be eliminated, and that can happen in one of two ways:
  (1) $v\ \&\ w \equiv v$ by SE-ISECTELIM1, given that $v <: w$. By lemma 4, v and w must either both be records, or must both be functions. If they are both records, then t reduces by lemma 7. If they are both functions, then t reduces by E-COMPOSEFUN.
  (2) If $v\ \&\ w \equiv w'$ by SE-COMPOSEFUN/REC, then $v\ \&\ w \longrightarrow w'$ by E-COMPOSEFUN/REC.
- Case W-COMPOSE and W-INHERIT —
  Similar to W-ISECT; the same lemmas apply.
- Case W-INTERFACE — $t = \uparrow v$. $\uparrow v$ can always be reduced by E-INTERFACEFUN/REC.

THEOREM 2 (Preservation).
If $\Gamma \vdash t$ wf, $t \longrightarrow t'$ then $\Gamma \vdash t'$ wf, $t \equiv t'$
Proof: By induction on the derivation of t wf.

- Case $t = E[u]$, and $u \longrightarrow u'$. By lemma 5, u wf. By the induction hypothesis, $u \equiv u'$ and u' wf. $E[u] \equiv E[u']$, and $E[u']$ wf by lemma 6.
- Case W-APPLY — $t = v(w)$. We know v must be a well-formed function. $v(w) \equiv t'$ by SE-APPLY. t' wf by lemma 3.
- Case W-PROJECT — $t = v.\ell$. We know v must be a well-formed record. $v.l \equiv t'$ by SE-PROJECT. t' wf by lemma 3.
- Case W-ISECT — $t = v\ \&\ w$. The intersection can be eliminated in one of two ways:
  (1) $v\ \&\ w \equiv v$ by SE-ISECTELIM1, given that $v <: w$. By lemma 4, v and w must either both be well-formed records, or must both be well-formed functions. If they are both records, then by lemma 7, $v\ \&\ w \longrightarrow t'$, such that $t' \equiv v$. Since v wf, it follows that t' wf.
  If v and w are functions, then $v\&w \equiv t'$ by SE-COMPOSEFUN. Moreover, the body of v must be a subtype of the body of w, so the intersection of the two bodies is well-formed.
  (2) $v\ \&\ w \equiv t'$ by SE-COMPOSEFUN/REC. We already have a proof that t' wf.
- Case W-COMPOSE and W-INHERIT —
  Similar to W-ISECT. In the case of record composition however, the previous line of reasoning shows that $\mu x\{\overline{c}\}\ \&*\ \mu x\{\overline{d}\} \longrightarrow \mu x\{\overline{e}\}$, such that $\mu x\{\uparrow\overline{e}\}$ wf (it only verifies the interface). Every field implementation in $\overline{e}$ is well-formed in the context of either $\mu x\{\overline{c}\}$ or $\mu x\{\overline{d}\}$, and is thus well-formed in the context of $\mu x\{\overline{e}\}$ by lemma 2.
- Case W-INTERFACE — $t = \uparrow v$. $\uparrow v \equiv t'$ by SE-INTERFACE.
  If v is a record, then d_ℓ wf implies $\uparrow d_\ell$ wf for all ℓ.
  If v is $\lambda(x : u).\ s$, then s wf implies $\uparrow s$ wf by W-INTERFACE.

6.9 Decidability and Recursion

THEOREM 3 (Decidability). The type system of DEEP is undecidable.

Proof: Evaluation of a term may not terminate because records allow general recursion. Subtyping is undecidable because subtyping includes evaluation. (By theorem 2, $t \longrightarrow t'$ implies that $t \equiv t'$.) Well-formedness depends on subtyping, so all judgments are undecidable. □

The fact that the type system is undecidable is not necessarily a problem, so long as a decidable semi-algorithm exists which captures the cases that we wish to model — namely systems of mutually recursive virtual classes. We present an informal outline of such an algorithm, but do not provide any proofs.

Our strategy uses fields to ascribe types to recursive implementations. Consider the following non-terminating definition:

```
μg{ inf = g.inf + 1; }
```

This record is not well-formed because `g.inf` has no bounding type — there is no value v for which `g.inf` $<: v$ — and it can therefor not be used as an argument to "+". When the compiler attempts to find such a bounding type, it will fail to terminate. This is an example of a *dependency cycle* — the bounding type for `g.inf` depends on `g.inf`.

Cycles can be broken by hiding recursive definitions within fields. The following definition is well-formed because `inf` is given an explicit upper bound of `Int`.

```
μg{ inf: Int = g.inf + 1; };
```

Surprisingly enough, regular recursive types such as lists and trees do not create cycles:

```
μg{
  IntList = {
    head :: Int;
```

```
    tail :: g.IntList;
  };
};
```

The above definition is well-formed because IntList is already defined as a value, and therefore does not need to be reduced. (The slots of a record use lazy evaluation). Nevertheless, it is possible to create an *infinite chain of dependencies* using a list-like structure:

```
μx{
  n: Int = 0;
  succ   = x &* { n: Int = x.n + 1; };
  last   = x.succ.last;
}
```

It is the definition of last, not the definition of succ, which fails to terminate. The definition of succ has an upper bound of $\uparrow x$, which can be simplified to a value. Once again, we break the chain by hiding the recursive definition behind a field:

```
... last: Int = x.succ.last; ...
```

6.9.1 Eliminating unsafe recursion

The above examples demonstrate that many programs for which type-checking fails to terminate can be transformed into well-formed programs by adding type annotations. A practical compiler should detect places where annotations are required (or might be required) and issue errors rather than stepping into an infinite loop. A practical compiler can be constructed with two restrictions:

- Recursive calls are not allowed in the interface of functions and records.
- It is not possible to project the implementation of a field during type-checking.

A close look at rule SE-PROJECT reveals that whenever a field is projected from a record, the type-checker has a choice. It can choose to project the range of the field, or it can choose to project the implementation. We force it to always project the range.

Detecting recursive calls is not unduly difficult. A cycle is created by a path $x.l_1...l_n$ which points to the slot that is currently being type-checked. To eliminate such cycles, we use an old trick from lazy languages. A slot is temporarily overwritten with a "black hole" value while it is being type-checked, and any term which partially evaluates to "black hole" is an error.

Detecting infinite chains requires a bit more subtlety, because recursive types (e.g. lists and trees) should remain legal. A *recursive path* is one which points to an enclosing record or function. Such paths are well-formed, and it is legal to use them as bounding types. This makes it possible to perform *nominal* subtyping within a set of mutually recursive classes. However, it is not possible to expand a recursive path to a value, which makes it illegal to call an enclosing function, or to inherit from an enclosing record.

By eliminating recursive calls in the interface, we guarantee that the partial evaluation of any term t will terminate to yield a value v, such that $t <: v$. Partial evaluation proceeds as normal, but only the ranges of fields are accessible, so a non-recursive bounding type is substituted for every potentially recursive call. This restricts the precision of the partial evaluation engine (see section 5.2), but it yields a terminating algorithm.

This technique also ensures that the subtyping and well-formedness judgments terminate. If recursive paths are not expanded, then recursive applications of "$\lhd$" or "wf" are always invoked on a smaller term.

7. Extensions

The most obvious limitation of DEEP is that the core calculus only supports simple overriding. It is not possible for a method m(...) in a subclass to call super.m(...), which is obviously a crippling restriction. We eliminate this restriction as follows.

Instead of defining fields with the syntax $l : t = u$, we use the syntax $l : t =_s u$. The term s is a programmer-defined operator of type $\lambda(x : t, y : t).\ t$, which mixes implementations together. Composition of fields then changes to the following:

$$(l : t =_s u)\ \&*\ (l : t =_s u') \longrightarrow (l : t =_s s(u)(u'))$$

This version of composition no longer replaces one implementation with the other. Instead, it mixes the two implementations together in some arbitrary manner which is defined by the operator s. As far as the type system is concerned, it doesn't matter what the field implementation is, so long as the implementation is a subtype of the range. There are several possible uses of this extension.

To implement the **super** keyword as defined in Java and C++, each method of type M is encoded as a field of type $\lambda(x : M).\ M$. This type represents a function which takes the super-class definition as its first argument. The mixin-operator s is the flip of function composition. A composition of classes C_1 &* ... &* C_n will produce a composition of methods $m_n \circ ... \circ m_1$. The composition of methods is then applied to some base definition, usually a no-op, to yield a single method of type M which chains calls back from subclass to superclass.

To implement mixin composition with linearization, each method of type M is encoded as a field which holds a list of functions of type $\lambda(x : M).M$. The mixin-operator s merges two lists together, maintaining relative order, but eliminating duplicates. Calling the method involves folding the list with the flip of function composition, and then applying the result to some base definition as before.

Other mixin schemes are possible. The exact details the schemes above don't matter so much as the idea that mixins do not need to be part of the core language, but can be offloaded to a library.

8. Related Work

The idea of unifying types and objects comes from the the Ohmu programming language [30], which was a predecessor to DEEP.

The vc calculus [28] is a formalization of the gbeta and Caesar languages, and was the first calculus to provide full support for virtual classes as attributes of objects. The definition of vc includes several complications that don't arise in DEEP. Vc is an imperative language with a mutable heap, and it performs linearization of mixins within the calculus. As explained in section 7, DEEP offloads linearization to a library. DEEP also follows a purely functional approach: any imperative constructs would have to be emulated in some way — e.g. with monads [52] or uniqueness types [5].

Unlike DEEP, the vc calculus is provably decidable, but it gains decidability at the cost of several restrictions. The type system of vc is purely nominal; it only handles dependent *path types*. It is unclear how structural types such as generics could be added to vc. DEEP supports full dependent types, with both structural and nominal typing.

Second, vc does not support final bindings. Final bindings are an important tool in many situations — arrays as defined in section 4 are a good example. However, there does not seem to be any way to add them to vc. The vc calculus avoids name clashes during inheritance by requiring that all slot names be unique. In real implementations this requirement is enforced by automatic name-mangling. This technique doesn't work with final bindings, because finalizing a slot when refining a virtual superclass will break any subclasses which attempt to override the same slot (see section 6.4). DEEP avoids this problem by using exact types, and by re-checking inherited declarations.

Third, vc restricts inheritance to classes within the same *family* — i.e. it is not possible to inherit from a class in a different object. DEEP has no such restriction.

The Tribe calculus is both simpler and somewhat more powerful than vc, although its decidability has not been proven [17]. Like DEEP, Tribe uses singleton types to enrich the subtype relation. Tribe also includes an intuitive notion of subtyping between families that DEEP lacks. In Tribe, $t.l <: u.l$ if $t <: u$, whereas in DEEP, $t.l <: u.l$ only if $t \equiv u$. This more flexible subtype rule is possible because every class in Tribe has a unique enclosing record, so there is a relative path from this to enclosing records. The tradeoff is that like vc, classes in Tribe can only inherit from other classes in the same object. Inheritance in DEEP can cross module boundaries, but subtyping is less flexible as a result.

Nystrom and Myer's Jx language is an extension of Java which supports the refinement of static nested classes [41]. Although Jx does not currently support virtual inner classes, it does use a dependent type system, so this restriction does not appear to be intrinsic. The most unusual aspect of Jx is the use of *prefix types* instead of path types. The J& language extends Jx with a form of deep mixin composition much like the one proposed in this paper, except that name clashes must be disambiguated by hand [42].

Bruce, Odersky, and Wadler have proposed a statically safe mechanism for refining groups of classes, but their mechanism does not scale further — there is no way to define groups of groups, and it is not possible to inherit from a virtual class. [14] The .FJ calculus developed by Igarashi and Saito is an extension of featherweight Java which allows the refinement of static nested classes. .FJ has a similar set of restrictions: it is not possible to inherit from a virtual class. [33] The Concord language developed by Jolley et. al. uses the idea of class groups, but it lifts the restriction on inheritance. [34]

Odersky's νObj calculus [43] provides a formal treatment of virtual types, and is the basis for the Scala programming language. However, although Scala supports virtual types, it does not support virtual classes. Instead, scala supports explicit self-types, which can be used to emulate virtual classes to some degree. Like Beta, however, it is not possible in Scala to inherit from a virtual class. Moreover, Scala does not perform deep mixin composition automatically; nested classes must be mixed by hand [55].

Duggan and Sourelis have proposed mixin modules as an extension to ML [23]. Mixin modules have the same fixpoint semantics as records in DEEP, and use a similar form of composition. However, there is no subtype relation between mixin modules, so it is a form of implementation inheritance only.

Bruce has done a great deal of work on match subtyping, and introduced the concept of exact types [13] [12]. Exact types provide many of the benefits of dependent types with a great deal less hassle, but they have not yet been used to support full virtual classes.

Mixin layers implement deep mixin composition using C++ templates [46]. Like features, this is essentially a generative approach, since templates are not type-checked prior to instantiation.

The expression problem has been discussed at length in many papers. Solutions can be found for gbeta [27], Java generics [50], Scala [55], and Hyper/J [44]. Lopez-Herrejon provides a comparison of generative approaches in [37].

Although most prototype languages are untyped, the Omega language by Gunther Blaschek is an exception [9]. The type system in Omega is nominal, and like Java, it uses a global class table, which makes it unsuitable for virtual classes.

8.1 Relation to typed λ-calculi

Subtyping, dependent types, singleton types, and intersection types have all been extensively studied in the literature. Subtyping has been added to Girard's System F and F_ω to yield System $F_{<:}$ and $F^\omega_{<:}$, respectively. Many object-oriented concepts can be translated into these two systems; subtyping provides a good model for inheritance.

System $F^\omega_{<:}$ is not quite powerful enough to handle self-types, but it can be extended with recursive types and F-bounded polymorphism [15]. This extension essentially forms the basis for Java generics, which have been explored as an existing language mechanism for class refinement [50]. Generics with F-bounded polymorphism can capture patterns of mutual recursion, but the number of parameters per class scales with the number of classes involved, which makes them unwieldy for complex collaborations.

8.1.1 Intersection Types

Intersection types have been used in a number of calculi. System $F^\omega_\wedge$ is an extension of System $F^\omega_{<:}$ which has been used to model OO languages with multiple inheritance [18]. It is important to note that intersection types in $F^\omega_\wedge$ are actually much stronger than those in DEEP.

In DEEP, an intersection is only well-formed if it can be reduced to some other term that is not an intersection. As a result, the type $(\lambda(\mathsf{x}\colon \mathsf{Int}).\ \mathsf{Int})\ \&\ (\lambda(\mathsf{x}\colon \mathsf{Float}).\ \mathsf{Float})$ is ill-formed. System $F^\omega_\wedge$ has no such restriction – the analogous type $\mathsf{Int} \to \mathsf{Int} \wedge \mathsf{Float} \to \mathsf{Float}$ is perfectly valid.

As a result, intersections in DEEP do not actually allow more terms to be typed; they are merely a convenient way to reuse code by merging declarations. This weaker form of intersection is sufficient to handle multiple inheritance, but it is provably less powerful. The restriction is a result of the fact that type intersections in DEEP can be applied to objects, and type safety requires that such intersections be reducible.

8.1.2 Dependently typed λ-calculi

The systems described above cannot be used to model classes which are attributes of objects. A further extension is necessary: dependent types. Igarashi and Pierce show that virtual types can be encoded in a variant of System $F^\omega_{<:}$ which has been extended with dependently typed records [31]. However, their extension does not handle mutually recursive classes, and its safety and decidability has not been shown.

Dependent types are powerful medicine, and it has proved difficult to combine them with other language mechanisms. Adding general recursion to the extended calculus of constructions instantly renders the entire system undecidable. Type equality depends on object equality, and object equality involves evaluation, so the proof of decidability in CC relies on strong normalization [20]. Recursive types are equally problematic because they can be used to write a fix-point operator, thus causing the same problem. The Cayenne language [4], which adds dependent types to Haskell, runs afoul of this problem, as does DEEP.

Combining dependent types and subtyping is also surprisingly tricky. A naive combination of the two in the style of System $F^\omega_{<:}$ introduces mutual dependencies between the typing, subtyping, and kinding judgments which makes the meta-theory very difficult. In System $\lambda P_{<:}$ Aspinall and Compagnoni circumvent this issue by carefully controlling the order in which proofs are done [3]. In System $\lambda C_{<:}$, Gang Chen avoids circularity by careful restructuring of the subtype rules [16]. Neither system supports bounded quantification (as found in System $F_{<:}$) because of meta-theoretic difficulties. Jan Zwanenburg has successfully added subtyping with bounded quantification to Pure Type Systems; he avoids the circularity by defining subtypes over pre-terms, rather than well-typed terms [56].

Singleton types and singleton kinds are closely related to dependent types [47]. Aspinall makes the crucial observation that in a system with singleton types, type judgments can be expressed as subtype judgments, and vice versa [2]. The DEEP calculus exploits

this relationship to conflate typing and subtyping, which results in a significantly simpler meta-theory.

8.2 Relation to partial evaluation

Partial evaluation is a promising technique, and there is a vast literature on the subject. The classic text by Neil Jones gives a broad overview of the field [35]. A more recent survey of current progress and unsolved problems can be found in [36].

The purpose of partial evaluation is to specialize a program with respect to some static input, in the hope that this will yield faster code. Specialization is done by shifting computations from run-time to compile-time. The Achilles heel of partial evaluation is the fact that the evaluator may not terminate — see [36] for details. For every function call, the evaluator must decide whether to expand (i.e. inline or evaluate) the call, or *residualize* the call, in which case the computation is deferred until run-time. Partial evaluators can be broadly classified into two groups with regard to how they make this decision.

Online evaluators make decisions "on the fly", based on internal heuristics. Offline evaluators perform a global *binding-time analysis* phase before performing any evaluation. This phase adds annotations to source code that describe which computations are static (compile-time), and which are dynamic (run-time).

The DEEP calculus employs a primitive form of online evaluation. It evaluates expressions on the fly, but does not employ the sophisticated heuristics found in successful real-world evaluators. In order to ensure termination, DEEP requires the programmer to annotate the source code by hand, adding bounding types that force the type system to generalize at specific points. (DEEP generalizes where an ordinary partial evaluator would residualize.)

The use of bounding types in DEEP bears a close resemblance to abstract interpretation, which is a general technique used for a wide variety of static analysis [21]. Like partial evaluators, abstract interpreters may fail to terminate. Abstract interpreters use a technique called *widening*, which discards enough information to ensure that the analysis terminates. There is an obvious parallel between widening and the use of bounding types in DEEP, but we have not pursued this issue further.

9. Conclusion

The DEEP calculus provides type-safe support for virtual classes. Objects, classes, and modules are all modeled as records, and records may be nested to any arbitrary depth. This means that objects may contain classes as attributes.

OO inheritance is emulated by composing records together; the composition of two records will merge definitions from both parents. Unlike ordinary inheritance, such merging performs a *deep mixin composition*, in which definitions with the same name are recursively composed. This is the same mechanism found in feature-oriented programming, and it supports the incremental refinement of large-scale class hierarchies.

The most innovative aspect of the DEEP calculus is the fact that it combines types and objects into a single construct, which we call a *prototype*. Every object denotes a type, and every type is a first-class object. Type safety relies on subtyping between prototypes, rather than typing.

Prototypes have two main advantages. First, prototypes combine three big ideas — dependent types, singleton types, and subtyping — into a unified whole. This mechanism supports a smooth spectrum of type information, from the very abstract (e.g. Object) down to the very detailed (e.g. singletons). Because types are unified with objects, types which depend on objects become a natural part of the language, rather than a strange and difficult construct.

Second, the subtype relation in DEEP incorporates ideas from the partial evaluation and abstract interpretation communities. The static type of a term may just be an upper bound. However, it may also be an exact value which represents the result of evaluating the term. The type system includes an interpreter which is capable of evaluating arbitrary expressions at compile-time. Types thus become a tool not just for catching errors, but also for code generation and optimization.

9.1 Limitations and Future Work

The design of DEEP is an attempt to unify ideas from several different branches of computer science. Currently, this unification represents the "lowest common denominator" of more advanced systems. It is thus important to note some of the limitations of DEEP.

9.1.1 Dependent types

Sophisticated dependent type systems, such as Luo's UTT [38], and Connor McBride's Epigram [40], support *inductive data types*. These data types are a limited form of recursive types which only allow structures of finite size. Primitive recursion over such data types yields total functions that are guaranteed to terminate, thus allowing dependent types and recursion to be combined in a decidable system. Interestingly enough, such functions can be used to write "type cast" operators that prove algebraic identities – such as the commutativity and associativity of addition.

This mechanism addresses the major weakness of DEEP with regard to dependent types, as described in section 4.5. Unfortunately, the current definition of DEEP relies on general recursion, so there is no way to construct proofs of this form.

9.1.2 Partial Evaluation

As mentioned earlier, real-world partial evaluators either perform binding-time analysis, or use sophisticated heuristics to guide the evaluation process. DEEP does neither, which cripples its ability to apply partial evaluation to any real-world problems.

However, it might be possible to apply inductive data types here as well. Dependent type systems with indexed data types provide exactly the sort of information that a partial evaluator needs — proofs of termination. In some respects, DEEP represents a first attempt at combining these ideas, but there is a great deal of work to be done.

Acknowlegements

We would like to acknowledge the help of Phil Wadler and Don Batory, who provided comments and discussion on earlier drafts of this paper. We would also like to acknowledge the support of MZA Associates Corporation, which funded this research.

References

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

[2] D. Aspinall. Subtyping with singleton types. *Eighth International Workshop on Computer Science Logic*, 1994.

[3] D. Aspinall and A. Compagnoni. Subtyping dependent types. *Proceedings of 11th Annual Symposium on Logic in Computer Science*, 1996.

[4] L. Augustsson. Cayenne - a language with dependent types. *International Conference on Functional Programming*, pages 239–250, 1998.

[5] E. Barendsen and S. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 1996.

[6] K. Barrett et al. A monotonic superclass linearization for dylan. *Proceedings of OOPSLA*, 1996.

[7] D. Batory, J. Liu, and J. Sarvela. Refinements and multi-dimensional separation of concerns. *ACM SIGSOFT*, 2003.

[8] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *Proceedings of ICSE*, 2003.

[9] G. Blaschek. Type-safe OOP with prototypes: the concepts of Omega. *Structured Programming*, 12(12):1–9, 1991.

[10] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* Ph.D. thesis, University of Utah, 1992.

[11] G. Bracha and W. Cook. Mixin-based inheritance. *OOPSLA*, 1990.

[12] K. Bruce. Some challenging typing issues in object-oriented languages. *Electronic Notes in Theoretical Computer Science 82 No. 8*, 2003.

[13] K. Bruce, A. Fiech, and L. Petersen. Subtyping is not a good "match" for object-oriented languages. *Proceedings of ECOOP*, 1997.

[14] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. *Proceedings of ECOOP*, 1998.

[15] P. Canning, W. Cook, W. Hill, and W. Olthoff. F-bounded polymorphism for object-oriented programming. *Proceedings of FPLCA*, 1989.

[16] G. Chen. Subtyping calculus of constructions (extended abstract). *Proceedings of the International Symposium on Mathematical Foundations of Computer Science*, 1997.

[17] D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad. Tribe: More types for virtual classes. *In submission*, 2005.

[18] A. Compagnoni and B. Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, 1996.

[19] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. *Proceedings of POPL*, 1990.

[20] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 1(2/3), 1988.

[21] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[22] D. Dreyer, R. Harper, and K. Crary. Toward a practical type theory for recursive modules. *Technical Report CMU-CS-01-112*, 2001.

[23] D. Duggan and C. Sourelis. Mixin modules. *Proceedings of the International Conference on Functional Programming*, 1996.

[24] E. Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance.* Ph.D. thesis. Department of Computer Science, University of Aarhus, Denmark, 1999.

[25] E. Ernst. Propagating class and method combination. *Proceedings of ECOOP*, 1999.

[26] E. Ernst. Family polymorphism. *Proceedings of ECOOP*, 2001.

[27] E. Ernst. Higher order hierarchies. *Proceedings of ECOOP*, 2003.

[28] E. Ernst, K. Ostermann, and W. Cook. A virtual class calculus. *Proceedings of POPL*, 2006.

[29] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. *ACM Symposium on Principles of Programming Languages*, 1998.

[30] D. Hutchins. The power of symmetry: Unifying inheritance and generative programming. *OOPSLA Companion, DDD Track*, 2003.

[31] A. Igarashi and B. Pierce. Foundations for virtual types. *Proceedings of ECOOP*, 1999.

[32] A. Igarashi, B. Pierce, and P. Wadler. Featherweight java, a minimal core calculus for java and gj. *Proceedings of OOPSLA*, 1999.

[33] A. Igarashi, C. Saito, and M. Viroli. Lightweight family polymorphism. *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, 2005.

[34] P. Jolly, S. Drossopoulou, C. Anderson, and K. Ostermann. Simple dependent types: Concord. *Proceedings of FTfJP*, 2005.

[35] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1993.

[36] N. D. Jones and A. J. Glenstrup. Program generation, termination, and binding-time analysis. *Proceedings of GPCE*, 2002.

[37] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technologies. *Proceedings of ECOOP*, 2005.

[38] Z. Luo. *Computation and reasoning: a type theory for computer science*. Oxford University Press, Inc., New York, NY, USA, 1994.

[39] O. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. *Proceedings of OOPSLA*, 1989.

[40] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[41] N. Nystrom, S. Chong, and A. Myers. Scalable extensibility via nested inheritance. *Proceedings of OOPSLA*, 2005.

[42] N. Nystrom, X. Qi, and A. Myers. J&: Nested intersection for scalable software composition. *Proceedings of OOPSLA*, 2006.

[43] M. Odersky, V. Cremet, C. Roeckl, and M. Zenger. A nominal theory of objects with dependent types. *Proceedings of ECOOP*, 2003.

[44] W. H. Peri Tarr, H. Ossher. N degrees of separation: Multi-dimensional separation of concerns. *Proceedings of ICSE*, 1999.

[45] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. *Proceedings of ECOOP*, 2002.

[46] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. *Proceedings of ECOOP*, 1998.

[47] C. Stone. *Singleton Kinds and Singleton Types*. PhD thesis, Carnegie Mellon University, 2000.

[48] K. K. Thorup and M. Torgersen. Unifying genericity — combining the benefits of virtual types and parameterized classes. *Proceedings of ECOOP*, 1999.

[49] M. Torgersen. Virtual types are statically safe. *5th Workshop on Foundations of Object-Oriented Languages*, 1998.

[50] M. Torgersen. The expression problem revisited. four new solutions using generics. *Proceedings of ECOOP*, 2004.

[51] D. Ungar and R. Smith. Self, the power of simplicity. *Proceedings of OOPSLA*, 1987.

[52] P. Wadler. The essence of functional programming. *Proceedings POPL*, 1992.

[53] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 2004.

[54] H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

[55] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. *Workshop on Foundations of Object-Oriented Languages*, 2005.

[56] J. Zwanenburg. Pure type systems with subtyping. *International Conference on Typed Lambda Calculi and Applications*, 1999.