Objects, Classes, Abstractions

Luca Cardelli

Digital Equipment Corporation
Systems Research Center

Based on joint work with Martín Abadi, ML2000 Committee discussions, and other relevant literature

FOOL'97

Convergence of O-O and Polymorphism

- Polymorphic languages want to be more object-oriented
  - ~ Quest (polymorphism + subtyping)
  - ~ Abel (polymorphism + F-bounded subtyping)
  - ~ Rapide (modules/polymorphism + F-bounded subtyping)
  - ~ ML2000 (modules/polymorphism + objects +? classes)
- Object-oriented languages want to be more polymorphic
  - ~ Modula-3 (modules + classes + templates)
  - ~ C++ (classes + templates)
  - ~ Java (classes +? templates)
- How can we make this work?

Reductionist Strategy

- Working hypothesis

  Smooth combination and integration of complex language features requires a good understanding of their typing properties.

- Strategy

  Try to explain complex ad-hoc features by less complex and more general features.

- Problems
  - ~ Very general features may be incompatible with each other.
  - ~ Combinations of orthogonal general features may fail to capture desired "invariants" of ad-hoc features.
- Results
  - ~ Has the reductionist strategy worked well so far?
  - ~ Will it always work?
- Cop-out
  - ~ Failed reductionism begets reductionism at a different level.

Objects, Classes, Abstractions

- Objects
  - ~ Reductionist strategy only partially successful.
    Better take object types as primitive after all.
    Problems in capturing structural invariants.
  - ~ Still, it inspired greater understanding and considerable simplifications.
  - ~ Neo-reductionism: take objects as primitive, but nothing else.
- Classes
  - ~ Reductionist strategy might be successful.
  - ~ It had better be.

- Abstractions
  - ~ Reductionism highly successful.
    (Abstractions ≈ Existentials ≈ Universals ≈ Polymorphism.)
- Objects + Abstraction
  (state/behavior control and encapsulation)
  - ~ Successful by a variety of different techniques. (Scoping, typing.)
- Classes + Abstraction
  (inheritance control and encapsulation)
  - ~ Open.

Outline

- Interpretations of objects
  - ~ Summary of various techniques.
- Interpretations of classes
  - ~ One particular basic technique.
- Interpretations of abstraction
  - ~ Brief summary of well-known material.
- Combining interpretations of classes and abstractions
  - ~ Difficulties and speculations.

Objects vs. Procedures

- Object-oriented programming languages have introduced (or popularized) a number of ideas and techniques.
- However, on a case-by-case basis, one can often emulate objects in procedural languages. Are object-oriented concepts reducible to procedural concepts?
  - ~ It is easy to emulate the operational semantics of objects.
  - ~ It is a little harder to emulate object types.
  - ~ It is much harder to emulate object types and their subtyping properties.
  - ~ In practice, this reduction is not feasible or attractive.

The Translation Problem

- N.B.: we deal with calculi as an approximation to what would happen in full-blown programming languages.
- The problem is to find a translation from an object calculus to a λ-calculus:
  - ~ The object calculus should be reasonably expressive.
  - ~ The λ-calculus should be standard enough.
  - ~ The translation should be faithful; in particular it should preserve subtyping.

- Aims:
  - ~ Provide a semantics that uses "ordinary" concepts.
  - ~ Provide an explanation of object typing.
  - ~ Suggest and validate reasoning principles for objects.
- Numerous attempts and techniques.

Notation

- Object notation, used informally:

$a,b ::=$	terms
x	variable
$[f_k{=}b_k\ ^{k\in 1..p} \mid l_i{=}\varsigma(x_i)b_i\ ^{i\in 1..n}]$	objects with fields & methods
$a_{\circ}l$	field selection
$a_{\circ}l{:=}b$	field update
$a.l$	method invocation
$a.l \Leftarrow \varsigma(x)b$	method update

  - ~ Plus typing annotation whenever convenient.
- Object type notation:

$$[f_k{:}B_k\ ^{k\in 1..p} \mid l_i{:}B_i\ ^{i\in 1..n}]$$

$$\mathrm{Obj}(X)\ [f_k{:}B_k\ ^{k\in 1..p} \mid l_i{:}B_i\{X\}\ ^{i\in 1..n}]$$

The Self-Application Semantics

- The self-application interpretation maps an object to a record of functions.
- On method invocation, the whole object is passed to the method as a parameter.

Untyped self-application interpretation

$[l_i{=}\varsigma(x_i)b_i\ ^{i\in 1..n}] \ \triangleq\ \langle l_i{=}\lambda(x_i)b_i\ ^{i\in 1..n}\rangle$	(l_i distinct)
$o.l_j \ \triangleq\ o{\cdot}l_j(o)$	($j\in 1..n$)
$o.l_j \Leftarrow \varsigma(y)b \ \triangleq\ o{\cdot}l_j{:=}\lambda(y)b$	($j\in 1..n$)

The Self-Application Semantics (Typed)

- A typed version is obtained by representing object types as recursive record types:

$$[l_i{:}B_i\ ^{i\in 1..n}] \ \triangleq\ \mu(X)\langle l_i{:}X\to B_i\ ^{i\in 1..n}\rangle$$

Self-application interpretation

$A \equiv [l_i{:}B_i\ ^{i\in 1..n}] \ \triangleq$ $\mu(X)\langle l_i{:}X\to B_i\ ^{i\in 1..n}\rangle$	(l_i distinct)
$[l_i{=}\varsigma(x_i{:}A)b_i\ ^{i\in 1..n}] \ \triangleq\ fold(A,\langle l_i{=}\lambda(x_i{:}A)b_i\ ^{i\in 1..n}\rangle)$	
$o.l_j \ \triangleq\ unfold(o)\cdot l_j(o)$	$(j\in 1..n)$
$o.l_j \Leftarrow\varsigma(y{:}A)b \ \triangleq\ fold(A, unfold(o)\cdot l_j{:=}\lambda(y{:}A)b)$	$(j\in 1..n)$

- Unfortunately, the subtyping rule for object types fails to hold: a contravariant X occurs in all method types.

The State-Application Semantics (Typed)

- The state of an object, represented by a collection of fields *st*, is hidden by existential abstraction, so external updates are not possible.
  - ~ The troublesome method argument types are hidden as well, so this interpretation yields the desired subtypings.

$$[l_i{:}B_i\ ^{i\in 1..n}] \ \triangleq\ \exists(X)\ \langle st{:}\ X,\ mt{:}\ \langle l_i{:}X\to B_i\ ^{i\in 1..n}\rangle\rangle$$

  - ~ In the general case, code generation is driven by types (i.e. it is not syntax-directed).
  - ~ The typed translation is technically elegant, but in practice must be automated.
  - ~ It accounts well for class-based languages where methods are separate from fields, and where there is no method update.

The Split-Method Semantics (Typed)

$$[l_i{:}B_i\ ^{i\in 1..n}] \ \triangleq$$
$$\mu(Y)\ \exists(X{<:}Y)\ \langle r{:}X,\ l_i^{sel}{:}X\to B_i\ ^{i\in 1..n},\ l_i^{upd}{:}(X\to B_i)\to X\ ^{i\in 1..n}\rangle$$

- Has great properties
  - ~ We obtain both the expected semantics and the expected subtyping properties.
  - ~ The definition of the interpretation is syntax-directed.
  - ~ The interpretation covers method update. It extends naturally to other constructs: variance annotations, Self types (with some twists), imperative update, imperative cloning.
- But, clearly, cannot be used directly.

Summary of Object Encodings

- Some interpretations are good enough to explain objects in reasonable detail. But they require very advanced type systems and are elaborate.
- Although they are intellectually satisfying, they are not a practical replacement for primitive objects in programming languages.
- They suggest particularly simple object systems, akin to the ones found in object-based languages rather than those found in class-based languages.

How to Understand Classes?

- Many styles of interpretation are possible.
- We consider an interpretation that builds on the previous study of objects.
- The same kind of interpretation can be layered on top of module structures, instead of object structures.
- Initially, we do not consider abstraction/hiding/inheritance-control.

Review: Objects and Object Types

- Objects are packages of data (*instance variables*) and code (*methods*).
- Object types describe the shape of objects.

```
ObjectType CellType;
    var contents: Integer;
    method get(): Integer;
    method set(n: Integer);
end;

object cell: CellType;
    var contents: Integer := 0;
    method get(): Integer; return self.contents end;
    method set(n: Integer); self.contents := n end;
end;
```

where $a : A$ means that the program a has type A. So, *cell* : *CellType*.

Review: Classes

- Classes are ways of describing and generating collections of objects.

```
class cellClass for CellType;
    var contents: Integer := 0;
    method get(): Integer; return self.contents end;
    method set(n: Integer); self.contents := n end;
end;
var cell : CellType := new cellClass;
procedure double(aCell: CellType);
    aCell.set(2 * aCell.get());
end;
```


Review: Subclasses

- Subclasses are ways of describing classes incrementally, reusing code.

```
ObjectType ReCellType;
    var contents: Integer;
    var backup: Integer;
    method get(): Integer;
    method set(n: Integer);
    method restore();
end;

subclass reCellClass of cellClass for ReCellType;          (Inherited:
    var backup: Integer := 0;                                 var contents
    override set(n: Integer);                                 method get)
        self.backup := self.contents;
        super.set(n);
    end;
    method restore(); self.contents := self.backup end;
end;
```

Review: Subtyping and Subsumption

- Subtyping relation, $A <: B$

  An object type is a subtype of any object type with fewer components.

  (e.g.: *ReCellType* $<:$ *CellType*)

- Subsumption rule

  if $a : A$ and $A <: B$ then $a : B$

  (e.g.: *reCell* : *CellType*)

- Subclass rule

  cClass can be a subclass of *dClass* only if *cType* $<:$ *dType*

  (e.g.: *reCellClass* can indeed be declared as a subclass of *cellClass*)

An Interpretation of Classes

- Inheritance is method reuse.
  - ~ But one cannot reuse methods of existing objects: method extraction is not type-sound in typed languages.
  - ~ Therefore, we need classes, in addition to objects, to achieve inheritance. (Or delegation...)
- A *pre-method* is a function that is later used as a method.
- A class is a collection of pre-methods plus a way of generating new objects.

Classes as Objects

- A class is an object with:
  - ~ a *new* method, for generating new objects,
  - ~ code for methods for the objects generated from the class.
- For generating the object:

$$o \triangleq [l_i = \varsigma(x_i)\ b_i \ ^{i \in 1..n}]$$

we use the class:

$$\begin{aligned} c \triangleq\ & [new = \varsigma(z)\ [l_i = \varsigma(x)\ z.l_i(x) \ ^{i \in 1..n}], \\ & \quad l_i = \lambda(x_i)\ b_i \ ^{i \in 1..n}] \end{aligned}$$

  - ~ The method *new* is a **generator**. The call *c.new* yields *o*.
  - ~ Each field l_i is a **pre-method**.

Ex.: A Class for Cells

- Consider the object:

$$\begin{aligned} cell \triangleq\ & [contents = 0, \\ & \ set = \varsigma(x)\ \lambda(n)\ x.contents := n] \end{aligned}$$

- We obtain the class code:

$$\begin{aligned} & cellClass \triangleq \\ & \quad [new = \varsigma(z)\ [contents = \varsigma(x)\ z.contents(x),\ set = \varsigma(x)\ z.set(x)], \\ & \quad\ contents = \lambda(x)\ 0, \\ & \quad\ set = \lambda(x)\ \lambda(n)\ x.contents := n] \end{aligned}$$

  - ~ Writing the *new* method is tedious but straightforward.
  - ~ Writing the pre-methods is like writing the corresponding methods.
  - ~ *cellClass.new* yields a standard cell:

$$[contents = 0,\ set = \varsigma(x)\ \lambda(n)\ x.contents := n]$$

Inheritance

- Inheritance is the reuse of pre-methods.
  - Given a class c with pre-methods $c.l_i \; {}^{i \in 1..n}$ we may define a new class c':

    $$c' \triangleq [new=..., l_i=c.l_i \; {}^{i \in 1..n}, l_j=... \; {}^{j \in n+1..m}]$$

    We may say that c' is a subclass of c.
- Multiple inheritance is no sweat.

Ex.: Inheritance for Cells

- Consider a subclass of cell with "undo".
- We obtain the subclass code:

$$\begin{aligned} &uncellClass \triangleq \\ &\quad [new = \varsigma(z)\,[...], \\ &\quad\; contents = cellClass.contents, \\ &\quad\; set = \lambda(x)\; cellClass.set(x.undo := x), \\ &\quad\; undo = \lambda(x)\; x] \end{aligned}$$

  - The pre-method *contents* is inherited.
  - The pre-method *set* is overridden, though using a call to **super**.
  - The pre-method *undo* is added.

Object Types

- An **object type**

  $[l_i{:}B_i \; {}^{i \in 1..n}]$

  is the type of those objects with methods l_i, with a self parameter of type $A <: [l_i{:}B_i \; {}^{i \in 1..n}]$ and a result of type B_i.
- An object type with more methods is a **subtype** of one with fewer methods:

  $$[l_i{:}B_i \; {}^{i \in 1..n+m}] \quad <: \quad [l_i{:}B_i \; {}^{i \in 1..n}]$$

- Properties of object types:
  - Object types are **invariant** (not covariant, not contravariant) in their components.
  - An object can be used in place of another object with fewer methods, by **subsumption**:

    $$a : A \quad \wedge \quad A <: B \quad \Rightarrow \quad a : B$$

  - Subsumption is the basis for object-style polymorphism, and useful for inheritance:

    $$f : B \rightarrow C \quad \wedge \quad a : A \quad \wedge \quad A <: B \quad \Rightarrow \quad f(a) : C$$

    f implements l in B $\wedge$ $A <: B$ $\Rightarrow$
    f can implement l in A

Classes, with Typing

- If $A \equiv [l_i{:}B_i\ ^{i\in 1..n}]$ is an object type, then $Class(A)$ is the type of the classes for objects of type A:

$$Class(A) \triangleq [new{:}A,\ l_i{:}A \to B_i\ ^{i\in 1..n}]$$

$new{:}A$	is a **generator** for objects of type A.
$l_i{:}A \to B_i$	is a **pre-method** for objects of type A.

$$c : Class(A) \triangleq$$
$$[new = \varsigma(z{:}Class(A))\ [l_i = \varsigma(x{:}A)\ z.l_i(x)\ ^{i\in 1..n}],$$
$$l_i = \lambda(x_i{:}A)\ b_i\{x_i\}\ ^{i\in 1..n}]$$

$$c.new : A$$

~ Types are distinct from classes.

~ More than one class may generate objects of a type.

Inheritance, with Typing

- Inheritance is well-typed.

~ Let $A \equiv [l_i{:}B_i\ ^{i\in 1..n}]$ and $A' \equiv [l_i{:}B_i\ ^{i\in 1..n}, l_j{:}B_j\ ^{j\in n+1..m}]$, with $A' <: A$.

~ Note that $Class(A)$ and $Class(A')$ are not related by subtyping. Nor they need to be.

~ Let c: $Class(A)$, then for $i\in 1..n$

$$c.l_i{:}\ A \to B_i <: A' \to B_i.$$

Hence $c.l_i$ is a good pre-method for a class of type $Class(A')$.

~ We may now define a subclass c' of c:

$$c' : Class(A') \triangleq$$
$$[new=\ldots,\ l_i{=}c.l_i\ ^{i\in 1..n},\ l_j{=}\ldots\ ^{j\in n+1..m}]$$

where class c' inherits the methods l_i from class c.

~ So inheritance typechecks:

If $A'{<:}A$ then a class for A' may inherit from a class for A.

Ex.: Class Types for Cells

$$Class(Cell) \triangleq$$
$$[new : Cell,$$
$$contents : Cell \to Nat,$$
$$set : Cell \to Nat \to []]$$

$$Class(GCell) \triangleq$$
$$[new : GCell,$$
$$contents : GCell \to Nat,$$
$$set : GCell \to Nat \to [],$$
$$get : GCell \to Nat]$$

- $Class(GCell) <: Class(Cell)$ does not hold, but inheritance is possible:

$$Cell \to Nat <: GCell \to Nat$$
$$Cell \to Nat \to [] <: GCell \to Nat \to []$$

Variance Annotations

- Aim: finer control on field/method usage and on pre-method reuse.
  - ~ In order to gain expressiveness in a simple way (without resorting to quantifiers) we extend the syntax of object types with variance annotations:

    $[l_i \upsilon_i : B_i \ ^{i \in 1..n}]$

    Each υ_i is a variance annotation; it is one of ^{o}, $^+$, and $^-$.

- Intuitively,
  - ~ $^+$ means *read-only*: it prevents update, but allows covariant component subtyping;
  - ~ $^-$ means *write-only*: it prevents invocation, but allows contravariant component subtyping;
  - ~ ^{o} means *read-write*: it allows both invocation and update, but requires exact matching in subtyping.
  - ~ By convention, any omitted annotations are taken to be equal to ^{o}.

Subtyping with Variance Annotations

$[\ldots\ l^o{:}B\ \ldots] <: [\ldots\ l^o{:}B'\ \ldots]$	if $B \equiv B'$	invariant (read-write)
$[\ldots\ l^+{:}B\ \ldots] <: [\ldots\ l^+{:}B'\ \ldots]$	if $B <: B'$	covariant (read-only)
$[\ldots\ l^-{:}B\ \ldots] <: [\ldots\ l^-{:}B'\ \ldots]$	if $B' <: B$	contravariant (write-only)
$[\ldots\ l^o{:}B\ \ldots] <: [\ldots\ l^+{:}B'\ \ldots]$	if $B <: B'$	invariant <: variant
$[\ldots\ l^o{:}B\ \ldots] <: [\ldots\ l^-{:}B'\ \ldots]$	if $B' <: B$	

Protection for Objects

- Variance annotations can provide protection against updates from the outside. In addition, object components can be hidden by subsumption.

  Let $GCell \triangleq [contents : Nat, set : Nat \to [], get : Nat]$
  $PGCell \triangleq [set : Nat \to [], get : Nat]$
  $ProtectedGCell \triangleq [set^+ : Nat \to [], get^+ : Nat]$
  $gcell : GCell$
  then $GCell <: PGCell <: ProtectedGCell$
  so $gcell : ProtectedGCell$.

  - ~ Given a *ProtectedGCell*, one cannot access its *contents* directly.
  - ~ From the inside, *set* and *get* can still update and read *contents*.

Protection for Classes

- Using subtyping, we can provide protection for classes.
- We may associate two separate interfaces with a class type:
  - ~ The first interface is the collection of methods that are available in instances.
  - ~ The second interface is the collection of methods that can be inherited in subclasses.
- For an object type $A \equiv [l_i{:}B_i\ ^{i\in I}]$ with methods $l_i\ ^{i\in I}$ we consider:
  - ~ a restricted *instance interface*, determined by a set $Ins \subseteq I$, and
  - ~ a restricted *subclass interface*, determined by a set $Sub \subseteq I$.

- For an object type $A \equiv [l_i{:}B_i\ ^{i\in I}]$, and Ins, $Sub \subseteq I$, we define:

$$Class(A)_{Ins,Sub} \triangleq [new^{+}{:}[l_i{:}B_i\ ^{i\in Ins}],\ l_i{:}A\to B_i\ ^{i\in Sub}]$$

  - ~ $Class(A) <: Class(A)_{Ins,Sub}$ holds, so we get protection by subsumption.

- Particular values of *Ins* and *Sub* correspond to common situations.

$c : Class(A)_{\emptyset,Sub}$	is an abstract class based on A
$c : Class(A)_{Ins,\emptyset}$	is a leaf class based on A
$c : Class(A)_{I,I}$	is a concrete class based on A
$c : Class(A)_{Pub,Pub}$	has public methods $l_i\ ^{i\in Pub}$ and private methods $l_i\ ^{i\in I-Pub}$
$c : Class(A)_{Pub,Pub\cup Pro}$	has public methods $l_i\ ^{i\in Pub}$, protected methods $l_i\ ^{i\in Pro}$, and private methods $l_i\ ^{i\in I-Pub\cup Pro}$

Classes and Self

- As before, we associate a class type $Class(A)$ with each object type A.

$$A \equiv Obj(X)[l_i\upsilon_i{:}B_i\{X\}\ ^{i\in 1..n}]$$

$$Class(A) \triangleq [new{:}A,\ l_i{:}\forall(X<:A)X\to B_i\{X\}\ ^{i\in 1..n}]$$

$$c : Class(A) \triangleq [new=\varsigma(z{:}Class(A))\ obj(X=A)[l_i=\varsigma(s{:}X)z.l_i(X)(s)\ ^{i\in 1..n}],\ l_i = \lambda(Self<:A)\ \lambda(s{:}Self)\ \ldots\ ^{i\in 1..n}]$$

- Now pre-methods have polymorphic types.

Interpretations of Abstraction

- Untyped abstractions (value visibility).
  - ~ Scoping restrictions (static).
  - ~ Access restrictions (dynamic).
- Typed abstractions (type visibility).
  - ~ Restricted "views", e.g. subtyping, variance annotations.
  - ~ Representation hiding (ADT's).
  - ~ Partial representation hiding (combining the previous two).

The Bounded Existential Quantifier

- A natural candidate for flexible abstraction.
- The existentially quantified type $\exists(X<:A)B\{X\}$ is the type of the pairs $\langle A',b\rangle$ where A' is a subtype of A and b is a term of type $B\{\!\{A'\}\!\}$.
  - ~ The type $\exists(X<:A)B\{X\}$ can be seen as a partially abstract data type with *interface* $B\{X\}$ and with *representation type* X known only to be a subtype of A.
  - ~ It is partially abstract in that it gives some information about the representation type, namely, a bound.
- The pair $\langle A',b\rangle$ describes an element of the partially abstract data type with representation type A' and *implementation* b.

Object-Oriented Abstractions

- The famous "state encapsulation" property of objects is achieved mostly by value visibility restrictions (e.g. in untyped languages). Just as in closures.
- The more sophisticated "private" and "protected" properties of classes are also fairly simple value visibility restrictions that can be handled by restricting visibility.
- There is also a strong desire to use type visibility restrictions, e.g. to hide the representation of classes while still allowing extensions. This is where abstraction and classes start interfering in interesting ways.

Classes are not Abstract

- Classes are not abstractions. Classes are *raw code* that nobody should *ever* look at (contrary to common practice). They are the equivalent of values or modules, not of types or interfaces.
- Central question: how to combine abstraction with inheritance? Desired consequences:
  - ~ Representation hiding for classes.
  - ~ Modeling "final methods" and "final classes".
  - ~ Abstract hierarchies.
  - ~ Inheritance from abstracted classes.
  - ~ Creation of elements of abstracted classes.

Possible Approaches

- Abstraction first
  - ~ Put classes inside of modules (as in Modula-3). Classes provide inheritance, modules/interfaces provide abstraction.
  - ~ Unfortunately, standard modules are not extensible.
- Inheritability first
  - ~ There is a lot of momentum towards classes taking the role of modules.
  - ~ Therefore we should devise “class interfaces” that provide abstraction in addition to inheritability. (As opposed to “object interfaces” that just describe objects.)

Technical Problems

- Modeling final things
  - ~ Type systems do not distinguish between different values of the same type.
  - ~ But some concepts, such as “final method” are based on fixing a certain value.
  - ~ Since classes are value, “final classes” exhibit the same problem.
  - ~ There is hope though, since abstraction can be used to control the creation of values.

- Enforcing abstraction
  - ~ If we take an interpretation of classes, e.g.:

    $$Class(A) \;\equiv\; [new{:}A,\ l_i : A \to B_i\ ^{i \in 1..n}]$$

    where exactly do we sprinkle the abstractions?
  - ~ It might seem natural to abstract over the object type of a class:

    $$AbsClass(A) \;\equiv\; \exists(X{<:}A)\ [new{:}X,\ l_i : X \to B_i\ ^{i \in 1..n}]$$

    then, the l_i cannot be inherited.

    Moreover, consider $A'{<:}A$:

    $$AbsClass(A') \;\equiv\; \exists(Y{<:}A')\ [new{:}Y,\ l_i : Y \to B_i\ ^{i \in 1..n+m}]$$

    then, *new* cannot be defined from the previous *new*.

  - ~ One might give the pre-methods concrete types:

    $$\exists(X{<:}A)\ [new{:}X,\ l_i : A \to B_i\ ^{i \in 1..n}]$$

    then, the pre-methods cannot use the (full) representation.

Conclusions

- We should have better type-theoretical understanding of O-O constructs. (Remember the working hypothesis.)
  - ~ Object encodings have been thrashed around quite a bit.
  - ~ Class encodings have still a long way to go, especially if we want to account for advanced features.
- Interactions of classes and abstraction are still mysterious, both in programming practice and in theory.
  - ~ There has always been a tension between inheritance and abstraction: classes are commonly used as *leaky ADT*'s.
  - ~ Is this conflict hopeless? Foundational studies should help bring this question into focus.