How to Tell a Compiler What We Think We Know?

Guy L. Steele Jr.
Software Architect, Oracle Labs

SPLASH-I Keynote
Friday, November 4, 2016

ORACLE®

We Begin with a Digression

Why do these slides have a strange aspect ratio?

If my slides are 4:3 but the projector is 16:9, 25% of the screen is wasted:

And if my slides are 16:9 but the projector is 4:3, 25% of the screen is wasted:

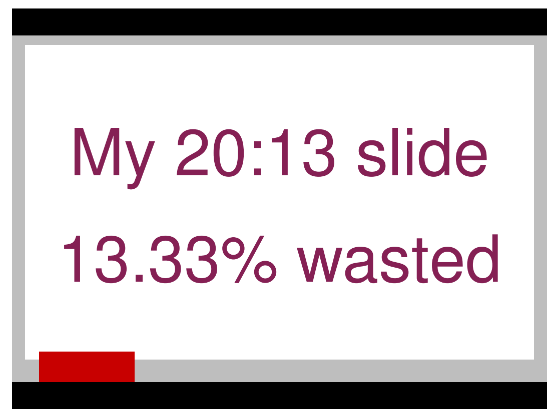
ORACLE

These Slides Have a 20:13 Aspect Ratio

The *optimal* compromise ratio is $8{:}3\sqrt{3} = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{5 + \cfrac{1}{1 + \cfrac{1}{4 + \cdots}}}}}$.

Sucessive truncations of this continued fraction produce approximants 1:1, 2:1, 3:2, 17:11, 20:13, 97:63, . . .

Using 20:13 with either 16:9 or 4:3 projection, less than 13.5% is wasted:



You may want to give this a try. If you can't be bothered with 20:13, try 3:2 —at least until 16:9 projectors become ubiquitous.

(End of digression.)

An Offhand Quote

If it’s worth telling another programmer,
it’s worth telling the compiler, I think.

—Guy Steele, in *Coders at Work* by Peter Seibel (2009)

ORACLE®

A Modest Internet Meme

I’m always delighted by the light touch and stillness of early programming languages. Not much text; a lot gets done. Old programs read like quiet conversations between a well-spoken research worker and a well-studied mechanical colleague, not as a debate with a compiler. Who’d have guessed sophistication bought such noise.

—Richard P. Gabriel, in *50 in 50* (2007)

https://vimeo.com/25958308 at 38:40

Google Search: 222 hits

ORACLE

Example Javadoc Comment #1

```
interface java.util.Collection<E>
 boolean add(E e)
```

...

If a collection refuses to add a particular element for any reason other than that it already contains the element, it *must* throw an exception (rather than returning `false`). This preserves the invariant that a collection always contains the specified element after this call returns.

Example Javadoc Comment #2

```
interface java.math.BigInteger
 public BigInteger shiftLeft(int n)
```

Returns a `BigInteger` whose value is `(this << n)`.
The shift distance, `n`, may be negative, in which case this method performs a right shift. (Computes `floor(this *` 2^n`)`.)

Example Javadoc Comment #3

```
interface java.lang.Math
```

```
public static float scalb(float f, int scaleFactor)
```

Returns $\texttt{f} \times 2^{\texttt{scaleFactor}}$ rounded as if performed by a single correctly rounded floating-point multiply to a member of the float value set.

Example Javadoc Comment #4

```
class java.lang.Object
 public int hashCode()
```

...

If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.

Example Javadoc Comment #5

```
class java.util.Vector<E>
 public int IndexOf(Object o)
```

...

Returns the index of the first occurrence of the specified element in this vector, or `-1` if this vector does not contain the element. More formally, returns the lowest index `i` such that `(o==null ? get(i)==null : o.equals(get(i)))`, or `-1` if there is no such index.

Example Javadoc Comment #6

```
class java.lang.Object
 public int equals(Object o)
```

The `equals` method implements an equivalence relation on non-null object references:

- It is reflexive: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is symmetric: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is transitive: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is consistent: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

ORACLE

Example Javadoc Comment #7

```
interface java.util.Collection<E>
```

```
public int equals(Object o)
```

... programmers who implement the `Collection` interface "directly" (in other words, create a class that is a `Collection` but is not a `Set` or a `List`) must exercise care if they choose to override the `Object.equals`. It is not necessary to do so, and the simplest course of action is to rely on `Object`'s implementation, but the implementor may wish to implement a "value comparison" in place of the default "reference comparison." (The `List` and `Set` interfaces mandate such value comparisons.)

The general contract for the `Object.equals` method states that equals must be symmetric (in other words, `a.equals(b)` if and only if `b.equals(a)`). The contracts for `List.equals` and `Set.equals` state that lists are only equal to other lists, and sets to other sets. Thus, a custom `equals` method for a collection class that implements neither the `List` nor `Set` interface must return `false` when this collection is compared to any list or set. (By the same logic, it is not possible to write a class that correctly implements both the `Set` and `List` interfaces.)

Example Javadoc Comment #8

```
class java.util.stream.Stream<T>

 T reduce(T identity,
          BinaryOperator<T> accumulator)
```

. . .

The `identity` value must be an identity for the `accumulator` function. This means that for all `t`, `accumulator.apply(identity, t)` is equal to `t`. The `accumulator` function must be an associative function.

Example Javadoc Comment #9

```
interface java.util.Collection<E>
```

```
default Stream<E> stream()
```

Returns a sequential `Stream` with this collection as its source.

This method should be overridden when the `spliterator()` method cannot return a spliterator that is `IMMUTABLE`, `CONCURRENT`, or late-binding.

Example Javadoc Comment #10

```
class java.util.regex.Matcher
```

...

Instances of this class are not safe for use by multiple concurrent threads.

Example Javadoc Comment #11

```
class java.util.concurrent.ConcurrentLinkedDeque<E>
```

. . .

Concurrent insertion, removal, and access operations execute safely across multiple threads.

. . .

Iterators and spliterators are *weakly consistent*.

Example Javadoc Comment #12

`interface java.awt.dnd.DropTargetListener`

`void drop(DropTargetDropEvent dtde)`

This method is responsible for undertaking the transfer of the data associated with the gesture. The `DropTargetDropEvent` provides a means to obtain a `Transferable` object that represents the data object(s) to be transfered.

From this method, the `DropTargetListener` shall accept or reject the drop via the `acceptDrop(int dropAction)` or `rejectDrop()` methods of the `DropTargetDropEvent` parameter.

Subsequent to `acceptDrop()`, but not before, `DropTargetDropEvent`'s `getTransferable()` method may be invoked, and data transfer may be performed via the returned `Transferable`'s `getTransferData()` method.

At the completion of a drop, an implementation of this method is required to signal the success/failure of the drop by passing an appropriate boolean to the `DropTargetDropEvent`'s `dropComplete(boolean success)` method.

ORACLE

What Is the Role of the Compiler (or IDE)?

- To translate code for machine execution
- To perform various optimizations
- To prevent “incorrect” programs from executing
  - Type-checking
  - Interfaces
  - Contracts
  - More generally, to verify certain claims by the programmer
- To report various properties of the program to the programmer
- To take directions from the programmer about how to carry out all of these activities

What Do We Want to Say?

Compilers contain specialized theorem provers (such as type analysis and flow analysis), and they are becoming somewhat more general.

What sort of claims would we like a compiler to verify?

How should we express such claims?

Will the claims themselves, in effect, become programs that need all the help and tools and abstractions of the base language?

ORACLE

Relationships and Contraints among Entities

- Various kinds of entities
  - Different data structures
  - Same data structure at different points in time
  - Different methods
  - One method and its arguments
- Various kind of relationships
  - Types; sources and sinks; invariants; temporal sequencing
- Expressed in various ways
  - "Plain English"; technical English
  - Chunks of code; mathematical notation

ORACLE

Attributes of a Single Function or Method

- Pure (free of side effects)
- Symmetric / commutative / antisymmetric
- Associative, idempotent
- Has an identity and/or a zero
- Injective (one-to-one) / surjective (covers entire range) / bijective
- Performance or algorithmic complexity

Relationships among Functions and Methods

- Distributive: $a \times (b + c) = (a \times b) + (a \times c)$
  - Less obvious example: $a + (b \ \mathtt{max}\ c) = (a + b)\ \mathtt{max}\ (a + c)$, an important characteristic of the *tropical semiring*, recently used to get practical parallel speedups on a class of optimization algorithms

    Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Efficient parallelization using rank convergence in dynamic programming algorithms. *CACM* 59, 10 (September 2016), 85–92. DOI: http://dx.doi.org/10.1145/2983553

- Must call f before calling g
- f produces an argument suitable for g
- g requires a value produced by f
- Homomorphisms (when $length$ maps strings to integers, in effect it also maps concatenation to integer addition—this is a monoid homomorphism)

Describing Relationships among Data Items

Often we simply refer to these as *invariants*.

- "$1 \leq i \leq 100$" or "$i < j$" or "$n = |a|$ for array a"
- "These arrays are all the same length"
- "This array is one element longer than that one"
- "This array can hold anything that one can"
- "This array contains all the same values as that one, *except* . . . "
- "i is a valid index for a"
- "i is the index of the first element of a that satisfies p"
- "m is a count of the elements of a that satisfy p"

ORACLE

Describing Attributes of Aggregate Data

- Sorted
- Has no duplicates
- Some field has no duplicates
  - For example, the keys of a map (regarded as a set of pairs)
  - More generally, $map(f, a)$ has no duplicates
- Is in “normal form”
- This tree is a heap (no node has a larger value than any of its descendants)
- The Red-Black tree property
- Monoid-cached trees (every node contains $reduce(f)$ of leaves below it)

ORACLE

Transformations on Programs

Sometimes we derive a program by starting with a simple working version and then transforming it:

- Changes of representation
- Loop unrolling, loop interchange, and loop fusion
- Deforestation (“recursion fusion”)
- Conversion to continuation-passing style
- Refactoring

We don’t yet have a good and well-accepted metalanguage for recording and replaying such transformations.

We do have tools for version control that record all the different versions of a file over time, but precious little in the way of tools that record, analyze, and report *relationships* between successive versions (other than simple `diff`).

ORACLE

These Are Very Rich Ideas

Associated with all these ideas is
a vast literature of theorems
and application techniques.

How can we begin to communicate them
to a compiler?

Baby steps, baby steps.

ORACLE®

Haskell Type Classes: Semiring

```
class Eq s => Semiring s where
  zero :: s
  one :: s
  (.+.) :: s -> s -> s
  (.*.) :: s -> s -> s
```

https://hackage.haskell.org/package/weighted-regexp-0.1.0.0/docs/Data-Semiring.html

ORACLE

Haskell Type Class Semiring Built on Monoid?

```
class Eq s, Monoid s => Semiring s where
  zero :: s
  one :: s
  (.+.) :: s -> s -> s
  (.*.) :: s -> s -> s
```

But there is a problem here . . .

Haskell Type Class Semiring Built on Monoid??

```
class Eq s, Monoid s (.+.) zero, Monoid s (.*.) one
      => Semiring s where
   zero :: s
   one :: s
   (.+.) :: s -> s -> s
   (.*.) :: s -> s -> s
```

Underlined part is not actually valid Haskell syntax!

Haskell Type Class Semiring Built on Monoid???

```
class Eq s, Monoid s (.+.) zero, Monoid s (.*.) one
      => Semiring s (.+.) zero (.*.) one where
  zero :: s
  one :: s
  (.+.) :: s -> s -> s
  (.*.) :: s -> s -> s
```

That is, .+. and zero and .*. and one must become bindable parameters.

Underlined part is not actually valid Haskell syntax!

ORACLE

Haskell Type Class `Semiring`: Comments

A semiring is an additive commutative monoid with identity `zero`:

```
        a .+. b  ==  b .+. a
     zero .+. a  ==  a
(a .+. b) .+. c  ==  a .+. (b .+. c)
```

A semiring is a multiplicative monoid with identity `one`:

```
      one .*. a  ==  a
      a .*. one  ==  a
(a .*. b) .*. c  ==  a .*. (b .*. c)
```

Multiplication distributes over addition:

```
a .*. (b .+. c)  ==  (a .*. b) .+. (a .*. c)
(a .+. b) .*. c  ==  (a .*. c) .+. (b .*. c)
```

`zero` annihilates a semiring with respect to multiplication:

```
     zero .*. a  ==  zero
     a .*. zero  ==  zero
```

https://hackage.haskell.org/package/weighted-regexp-0.1.0.0/docs/Data-Semiring.html

Organization versus Enforcement

Haskell type classes provide a way to *organize* such algebraic abstractions, but they do not *enforce* them.

Monads use the Haskell type system to enforce restrictions on access to data and ordering of operations, at the expense of single-threading the entire program (or the relevant parts of the program), but the algebraic monad laws that *every* monad should obey are not enforced upon the monads themselves; they are merely *documented*:

Instances of Monad should satisfy the following laws:

```
return a >>= k  =  k a
m >>= return  =  m
m >>= (x -> k x >>= h)  =  (m >>= k) >>= h
```

https://hackage.haskell.org/package/base-4.9.0.0/docs/Control-Monad.html

ORACLE

These Ideas Have Been in the Air for Nearly Three Decades

Wadler and Blott speculated when they first introduced type classes in 1988:

It is natural to think of adding assertions to the class declaration, specifying properties that each instance must satisfy:

```
class  Eq a  where
  (==)  ::  a -> a -> Bool
  %  (==) is an equivalence relation
class  Num a  where
  zero, one ::  a
  (+), (*)  ::  a -> a -> a
  negate    ::  a -> a
  %  (zero, one, (+), (*), negate) form a ring
```

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. Proc. 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, New York, 1988, 60–76. http://dx.doi.org/10.1145/75277.75283

It is valid for any proof to rely on these properties, so long as one proves that they hold for each instance declaration. Here the assertions have simply been written as comments; a more sophisticated system could perhaps verify or use such assertions.

Fortress Traits for Binary Predicates and Operators

$$\texttt{trait}\ \mathrm{BinaryPredicate}[\![T\ \texttt{extends}\ \mathrm{BinaryPredicate}[\![T,\sim]\!],\texttt{opr}\ \sim]\!]\ \texttt{comprises}\ T\ \texttt{extends}\ \mathrm{Any}$$
$$\quad\texttt{abstract opr}\ \sim(\texttt{self},\mathit{other}\colon T)\colon \mathrm{Boolean}$$
$$\texttt{end}$$

$$\texttt{trait}\ \mathrm{BinaryOperator}[\![T\ \texttt{extends}\ \mathrm{BinaryOperator}[\![T,\odot]\!],\texttt{opr}\ \odot]\!]\ \texttt{comprises}\ T\ \texttt{extends}\ \mathrm{Any}$$
$$\quad\texttt{abstract opr}\ \odot(\texttt{self},\mathit{other}\colon T)\colon T$$
$$\texttt{end}$$

$$\texttt{trait}\ \mathbb{Z}\ \texttt{extends}\ \{\ \mathrm{BinaryPredicate}[\![\mathbb{Z},=]\!],\mathrm{BinaryOperator}[\![\mathbb{Z},+]\!],\ldots\}$$
$$\quad\texttt{opr}\ =(\texttt{self},\mathit{other}\colon \mathbb{Z})\colon \mathrm{Boolean} = \ldots$$
$$\quad\texttt{opr}\ +(\texttt{self},\mathit{other}\colon \mathbb{Z})\colon \mathbb{Z} = \ldots$$
$$\quad\ldots$$
$$\texttt{end}$$

The "$\texttt{mixin}$" Abbreviation I Wish We Had Used

$\texttt{mixin}$ BinaryPredicate$[\![T, \texttt{opr}\ \sim]\!]$ $\texttt{extends}$ Any   ⊛ I wish!

    $\texttt{abstract opr} \sim(\texttt{self}, \mathit{other}{:}\, T){:}\ \text{Boolean}$

$\texttt{end}$

$\texttt{mixin}$ BinaryOperator$[\![T, \texttt{opr}\ \odot]\!]$ $\texttt{extends}$ Any   ⊛ In my dreams

    $\texttt{abstract opr} \odot(\texttt{self}, \mathit{other}{:}\, T){:}\ T$

$\texttt{end}$

$\texttt{trait}\ \mathbb{Z}\ \texttt{extends}\ \{\ \text{BinaryPredicate}[\![\mathbb{Z}, =]\!], \text{BinaryOperator}[\![\mathbb{Z}, +]\!], \ldots\ \}$

    $\texttt{opr} =(\texttt{self}, \mathit{other}{:}\, \mathbb{Z}){:}\ \text{Boolean} = \ldots$

    $\texttt{opr} +(\texttt{self}, \mathit{other}{:}\, \mathbb{Z}){:}\ \mathbb{Z} = \ldots$

    $\ldots$

$\texttt{end}$

ORACLE

Fortress Summation Method (Incomplete)

$$sum⟦U\ \texttt{extends}\ \mathrm{BinaryOperator}⟦U,+⟧⟧\,(x\colon \mathrm{List}⟦U⟧)\colon U =$$

$$\quad \texttt{if}\ x.empty\ \texttt{then}\ \ldots\ \texttt{else}\ x.first + sum(x.rest)\ \texttt{end}$$

(We could have used a loop, but this keeps the example simple.)

ORACLE

“Fortress”: Declared Characteristics of Predicates

$\texttt{mixin}$ Reflexive$[\![T, \texttt{opr}\ \sim]\!]$ $\texttt{extends}$ $\{$ BinaryPredicate$[\![T, \sim]\!]$ $\}$
    $\texttt{property}$ $\forall (a\colon T)\ (a \sim a)$
$\texttt{end}$

$\texttt{trait}$ Symmetric$[\![T, \texttt{opr}\ \sim]\!]$ $\texttt{extends}$ $\{$ BinaryPredicate$[\![T, \sim]\!]$ $\}$
    $\texttt{property}$ $\forall (a\colon T, b\colon T)\ (a \sim b) \leftrightarrow (b \sim a)$
$\texttt{end}$

$\texttt{trait}$ Transitive$[\![T, \texttt{opr}\ \sim]\!]$ $\texttt{extends}$ $\{$ BinaryPredicate$[\![T, \sim]\!]$ $\}$
    $\texttt{property}$ $\forall (a\colon T, b\colon T, c\colon T)\ \big((a \sim b) \wedge (b \sim c)\big) \rightarrow (a \sim c)$
$\texttt{end}$

ORACLE

“Fortress”: Equivalence Relations

$$\texttt{mixin}\ \mathrm{EquivalenceRelation}[\![T, \texttt{opr}\ \sim]\!]$$
$$\qquad\qquad \texttt{extends}\ \{\ \mathrm{Reflexive}[\![T, \sim]\!], \mathrm{Symmetric}[\![T, \sim]\!], \mathrm{Transitive}[\![T, \sim]\!]\ \}$$
$$\texttt{end}$$

$$\texttt{trait}\ \mathbb{Z}\ \texttt{extends}\ \{\ \mathrm{EquivalenceRelation}[\![\mathbb{Z}, =]\!], \mathrm{BinaryOperator}[\![\mathbb{Z}, +]\!], \ldots\ \}$$
$$\quad \texttt{opr}\ =(\texttt{self}, \mathit{other}\colon \mathbb{Z})\colon \mathrm{Boolean} = \ldots$$
$$\quad \texttt{opr}\ +(\texttt{self}, \mathit{other}\colon \mathbb{Z})\colon \mathbb{Z} = \ldots$$
$$\quad \ldots$$
$$\texttt{end}$$

ORACLE

“Fortress”: Associativity and Commutativity

$\texttt{mixin}\ \text{ApproximatelyAssociative}[\![T, \texttt{opr}\ \odot, \texttt{opr}\ \approx]\!]$
$\qquad\texttt{extends}\ \{\ \text{BinaryOperator}[\![T,\odot]\!], \text{Reflexive}[\![T,\approx]\!], \text{Symmetric}[\![T,\approx]\!]\ \}$
$\quad\texttt{property}\ \forall(a\colon T, b\colon T, c\colon T)\ \big((a \odot b) \odot c\big) \approx \big(a \odot (b \odot c)\big)$
$\texttt{end}$

$\texttt{mixin}\ \text{Associative}[\![T, \texttt{opr}\ \odot]\!]$
$\qquad\texttt{extends}\ \{\ \text{ApproximatelyAssociative}[\![T,\odot,=]\!], \text{EquivalenceRelation}[\![T,=]\!]\ \}$
$\texttt{end}$

$\texttt{mixin}\ \text{ApproximatelyCommutative}[\![T, \texttt{opr}\ \odot, \texttt{opr}\ \approx]\!]$
$\qquad\texttt{extends}\ \{\ \text{BinaryOperator}[\![T,\odot]\!], \text{Reflexive}[\![T,\approx]\!], \text{Symmetric}[\![T,\approx]\!]\ \}$
$\quad\texttt{property}\ \forall(a\colon T, b\colon T)\ (a \odot b) \approx (b \odot a)$
$\texttt{end}$

$\texttt{mixin}\ \text{Commutative}[\![T, \texttt{opr}\ \odot]\!]$
$\qquad\texttt{extends}\ \{\ \text{ApproximatelyCommutative}[\![T,\odot,=]\!], \text{EquivalenceRelation}[\![T,=]\!]\ \}$
$\texttt{end}$

"Fortress": Monoids (Associative Operators with Identity)

$$\texttt{mixin}\ \mathrm{Monoid}[\![T, \texttt{opr}\ \odot]\!]$$
$$\quad \texttt{extends}\ \{\ \mathrm{Associative}[\![T, \odot]\!]\ \}$$

⊛ Approximate cases omitted

$$\quad \texttt{where}\ \{\ T\ \texttt{coerces}\ \mathrm{Identity}[\![\odot]\!]\ \}$$
$$\texttt{end}$$

$$\texttt{trait}\ \mathrm{CommutativeMonoid}[\![T, \texttt{opr}\ \oplus]\!]$$
$$\quad \texttt{extends}\ \{\ \mathrm{Monoid}[\![T, \oplus]\!], \mathrm{Commutative}[\![T, \oplus]\!]\ \}$$

⊛ Approximate cases omitted

$$\quad \texttt{where}\ \{\ T\ \texttt{coerces}\ \mathrm{Identity}[\![\oplus]\!]\ \}$$
$$\texttt{end}$$

$$\texttt{trait}\ \mathbb{Z}\ \texttt{extends}\ \{\ \mathrm{EquivalenceRelation}[\![\mathbb{Z}, =]\!], \mathrm{CommutativeMonoid}[\![\mathbb{Z}, +]\!], \ldots\ \}$$
$$\quad \texttt{opr}\ =(\texttt{self}, \mathit{other}\colon \mathbb{Z})\colon \mathrm{Boolean} = \ldots$$
$$\quad \texttt{opr}\ +(\texttt{self}, \mathit{other}\colon \mathbb{Z})\colon \mathbb{Z} = \ldots$$
$$\quad \texttt{coercion}\ \big(x\colon \mathrm{Identity}[\![+]\!]\big) = 0$$
$$\quad \ldots$$
$$\texttt{end}$$

Fortress Summation Method (Complete)

$$sum\,[\![U\ \texttt{extends}\ \mathrm{Monoid}[\![U,+]\!]]\!]\,(x\colon \mathrm{List}[\![U]\!])\colon U =$$
$$\quad \texttt{if}\ x.empty\ \texttt{then}\ \mathrm{Identity}[\![+]\!]\ \texttt{else}\ x.first + sum(x.rest)\ \texttt{end}$$

ORACLE

Fortress Summation Method (Complete)

$$sum\,[\![U \texttt{ extends } \mathrm{Monoid}[\![U,+]\!]]\!]\,(x\colon \mathrm{List}[\![U]\!])\colon U =$$
$$\quad \texttt{if } x.empty \texttt{ then } \mathrm{Identity}[\![+]\!] \texttt{ else } x.first + sum(x.rest) \texttt{ end}$$

$$\texttt{object } \mathrm{Identity}[\![\texttt{opr } \odot]\!] \texttt{ end}$$

ORACLE

"Fortress": SemiRings and Rings

$$
\begin{array}{l}
\texttt{trait}\ \mathrm{SemiRing}[\![T, \texttt{opr}\ \oplus, \texttt{opr}\ \otimes]\!] \\
\qquad\quad \texttt{extends}\ \{\ \mathrm{CommutativeMonoid}[\![T, \oplus]\!], \mathrm{Monoid}[\![T, \otimes]\!], \\
\qquad\qquad\qquad \mathrm{Distributive}[\![T, \otimes, \oplus]\!], \mathrm{ZeroAnnihilation}[\![T, \otimes]\!]\ \} \\
\qquad\quad \texttt{where}\ \{\ T\ \texttt{coerces}\ \mathrm{Identity}[\![\oplus]\!], T\ \texttt{coerces}\ \mathrm{Identity}[\![\otimes]\!], T\ \texttt{coerces}\ \mathrm{Zero}[\![\otimes]\!]\ \} \\
\texttt{end}
\end{array}
$$

$$
\begin{array}{l}
\texttt{trait}\ \mathrm{Ring}[\![T, \texttt{opr}\ \oplus, \texttt{opr}\ \ominus, \texttt{opr}\ \otimes]\!] \\
\qquad\quad \texttt{extends}\ \{\ \mathrm{AbelianGroup}[\![T, \oplus, \ominus]\!], \mathrm{SemiRing}[\![T, \oplus, \otimes]\!]\ \} \\
\qquad\quad \texttt{where}\ \{\ T\ \texttt{coerces}\ \mathrm{Identity}[\![\oplus]\!], T\ \texttt{coerces}\ \mathrm{Identity}[\![\otimes]\!], T\ \texttt{coerces}\ \mathrm{Zero}[\![\otimes]\!]\ \} \\
\texttt{end}
\end{array}
$$

$$
\begin{array}{l}
\texttt{trait}\ \mathrm{CommutativeRing}[\![T, \texttt{opr}\ \oplus, \texttt{opr}\ \ominus, \texttt{opr}\ \otimes]\!] \\
\qquad\quad \texttt{extends}\ \{\ \mathrm{Ring}[\![T, \oplus, \ominus, \otimes]\!], \mathrm{CommutativeMonoid}[\![T, \otimes]\!]\ \} \\
\qquad\quad \texttt{where}\ \{\ T\ \texttt{coerces}\ \mathrm{Identity}[\![\oplus]\!], T\ \texttt{coerces}\ \mathrm{Identity}[\![\otimes]\!], T\ \texttt{coerces}\ \mathrm{Zero}[\![\otimes]\!]\ \} \\
\texttt{end}
\end{array}
$$

“Fortress”: Properties of Integers $\mathbb{Z}$

```
trait ℤ extends { EquivalenceRelation⟦ℤ, =⟧, CommutativeRing⟦ℤ, +, −, ×⟧,
                  TotalOrderOperators⟦ℤ, <, ≤, ≥, >, CMP⟧, ... }
    opr =(self, other: ℤ): Boolean = ...
    opr +(self, other: ℤ): ℤ = ...
    coercion (x: Identity⟦+⟧) = 0
    opr ×(self, other: ℤ): ℤ = ...
    coercion (x: Identity⟦×⟧) = 1
    coercion (x: Zero⟦×⟧) = 0
    opr <(self, other: ℤ): Boolean = ...
    opr ≤(self, other: ℤ): Boolean = ...
    opr ≥(self, other: ℤ): Boolean = ...
    opr >(self, other: ℤ): Boolean = ...
    opr CMP(self, other: ℤ): TotalComparison = ...
    ...
end
```

ORACLE

"Fortress": Boolean Algebras

$$
\begin{array}{l}
\texttt{trait}\ \text{Boolean}\ \texttt{extends}\ \{\,\text{EquivalenceRelation}[\![\text{Boolean}, =]\!],\\
\qquad\qquad\qquad\qquad\qquad\quad \text{BooleanAlgebra}[\![\text{Boolean}, \wedge, \vee, \sim, \oplus]\!], \ldots\}\\
\quad \texttt{opr}\ \wedge(\texttt{self}, \mathit{other}\colon \text{Boolean})\colon \text{Boolean} = \ldots\\
\quad \texttt{coercion}\ (x\colon \text{Identity}[\![\wedge]\!]) = \mathit{true}\\
\quad \texttt{coercion}\ (x\colon \text{Zero}[\![\wedge]\!]) = \mathit{false}\\
\quad \texttt{opr}\ \vee(\texttt{self}, \mathit{other}\colon \text{Boolean})\colon \text{Boolean} = \ldots\\
\quad \texttt{coercion}\ (x\colon \text{Identity}[\![\vee]\!]) = \mathit{false}\\
\quad \texttt{coercion}\ (x\colon \text{Zero}[\![\vee]\!]) = \mathit{true}\\
\quad \texttt{opr}\ \sim(\texttt{self})\colon \text{Boolean} = \ldots\\
\quad \texttt{opr}\ \oplus(\texttt{self}, \mathit{other}\colon \text{Boolean})\colon \text{Boolean} = \ldots\\
\quad \texttt{coercion}\ (x\colon \text{Identity}[\![\oplus]\!]) = \mathit{false}\\
\texttt{end}
\end{array}
$$

ORACLE

Advantages of This Approach

- Expressive (at least for “classical algebraic properties”)
- Modular
- Programmer can provide multiple implementations
  - Which to use can depend on declared data characteristics
    - ⋆ Reduction of an associative function can be parallelized, but the non-associative case can also be addressed
    - ⋆ Searching of a sorted array can use binary search instead of linear
    - ⋆ Polymorphic method dispatch supports automatic selection

ORACLE

Disadvantages of This Approach

- Data-centric (applied only to methods, not to global functions)
- Complete checking requires a theorem prover
- No mechanism for abstraction of "`where` $\{T$ `coerces ...`$\}$"
  - Such material had to be repeated over and over
  - Using getter methods and name parameters would have been better
- Could be overkill
  - Maybe parametric polymorphism is really needed only for collections
  - Likewise, maybe this stuff is really needed only for a limited set of algebraic properties that could just be built into the compiler?
- Does not capture temporal constraints (such as sequencing)
- Not always clear when it's doing you any good

ORACLE

Equivalence Relations in Coq (A Proof Assistant)

```
Class Reflexive (R : relation A) :=
  reflexivity : forall x : A, R x x.

Class Symmetric (R : relation A) :=
  symmetry : forall x y, R x y -> R y x.

Class Transitive (R : relation A) :=
  transitivity : forall x y z, R x y -> R y z -> R x z.

Class Equivalence (R : relation A) : Prop := {
  Equivalence_Reflexive :> Reflexive R ;
  Equivalence_Symmetric :> Symmetric R ;
  Equivalence_Transitive :> Transitive R }.
```

https://coq.inria.fr/library/Coq.Classes.RelationClasses.html

ORACLE

Semirings in Coq

```
Class SemiRing A {e: Equiv A}
      {plus: RingPlus A} {mult: RingMult A}
      {zero: RingZero A} {one: RingOne A} : Prop :=
  { semiring_mult_monoid :> CommutativeMonoid A (op:=mult)(unit:=one)
  ; semiring_plus_monoid :> CommutativeMonoid A (op:=plus)(unit:=zero)
  ; semiring_distr :> Distribute mult plus
  ; semiring_left_absorb :> LeftAbsorb mult zero }
```

Bas Spitters and Eelis van der Weegen. Type Classes for Mathematics in Type Theory. *Mathematical Structures in Computer Science* 21, 4 (August 2011), 795–825. http://dx.doi.org/10.1017/S0960129511000119

We Are Now Beginning to See These Technologies in Haskell

- Testing the declared properties of type classes using QuickCheck

  Johan Jeuring, Patrik Jansson, and Cláudio Amaral. Testing type class laws. Proc. 2012 Haskell Symposium. ACM, New York, 2012, 49–60. http://dx.doi.org/10.1145/2364506.2364514

- Using a theorem prover to prove declared properties of type classes

  Andrew Farmer, Neil Sculthorpe, and Andy Gill. Reasoning with the HERMIT: Tool support for equational reasoning on GHC Core programs. Proc. 2015 Haskell Symposium. ACM, New York, 2015, 23–34. DOI=http://dx.doi.org/10.1145/2804302.2804303

  Andreas Arvidsson, Moa Johansson, and Robin Touche. Proving type class laws for Haskell. Proc. 17th Symposium on Trends in Functional Programming, June 2016. https://tfp2016.org/papers/TFP_2016_paper_20.pdf

I can say many things to the compiler.

But will they be relevant?

If not, I will be wasting:

- My effort (writing them down)
- The compiler's effort (verifying and re-verifying them)

ORACLE

Hanabi: A Cooperative Card Game

- Each card has a color (red, blue, green, yellow, white) and a number (1, 2, 3, 4, 5).
- Except for the 5's, there is more than one card of each kind.
- Each player is dealt five cards (if 2 or 3 players) or four cards (if 4 or 5).
- *You must not look at your own hand!*
- Each player can see all *other* hands.
- Players must help each other to make correct plays.

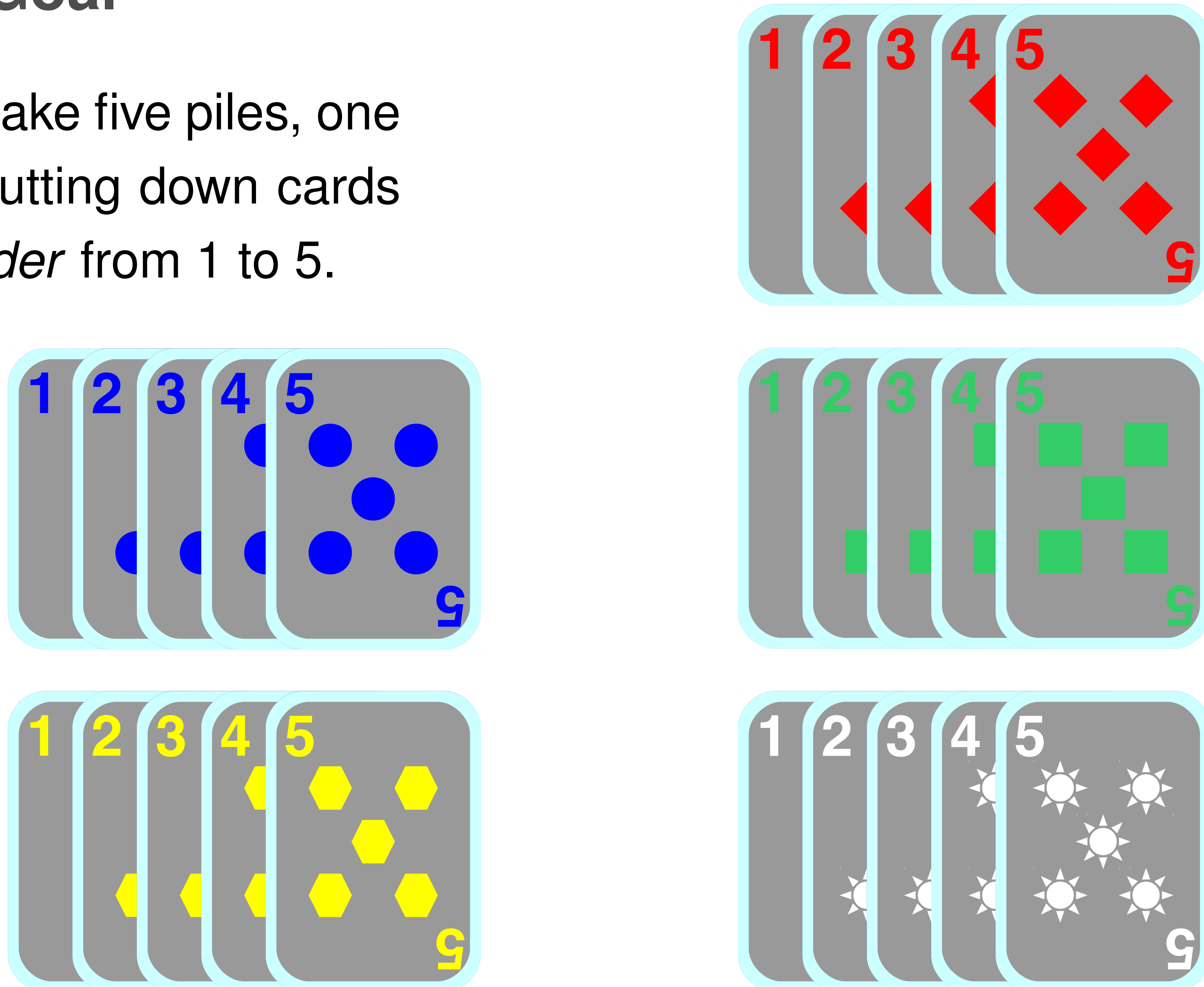
(Winner of *Spiel des Jahres* 2013. Available at Amazon or your friendly local game store.)

http://www.cocktailgames.com/en/game/hanabi/

ORACLE

Hanabi: The Goal

The goal is to make five piles, one of each color, putting down cards *in ascending order* from 1 to 5.



Hanabi: The Play (Simplified for This Talk)

Players take turns in the usual way in clockwise order.

On a turn, a player must choose to do exactly one of three things:

- Try to play one card (sight unseen!), then draw to replace
  - If the card cannot be played, it is a *mistake* (and is discarded)
  - On the third mistake, the game is lost
- Give information to another player:
  - Choose either a color or a number
  - Point out *every* card in that other player's hand that has that color or number
    - Or say, "You have no blue cards," "You have no 2's," etc.
- Discard one card, then draw to replace

ORACLE

Hanabi: The 60 Possible Utterances

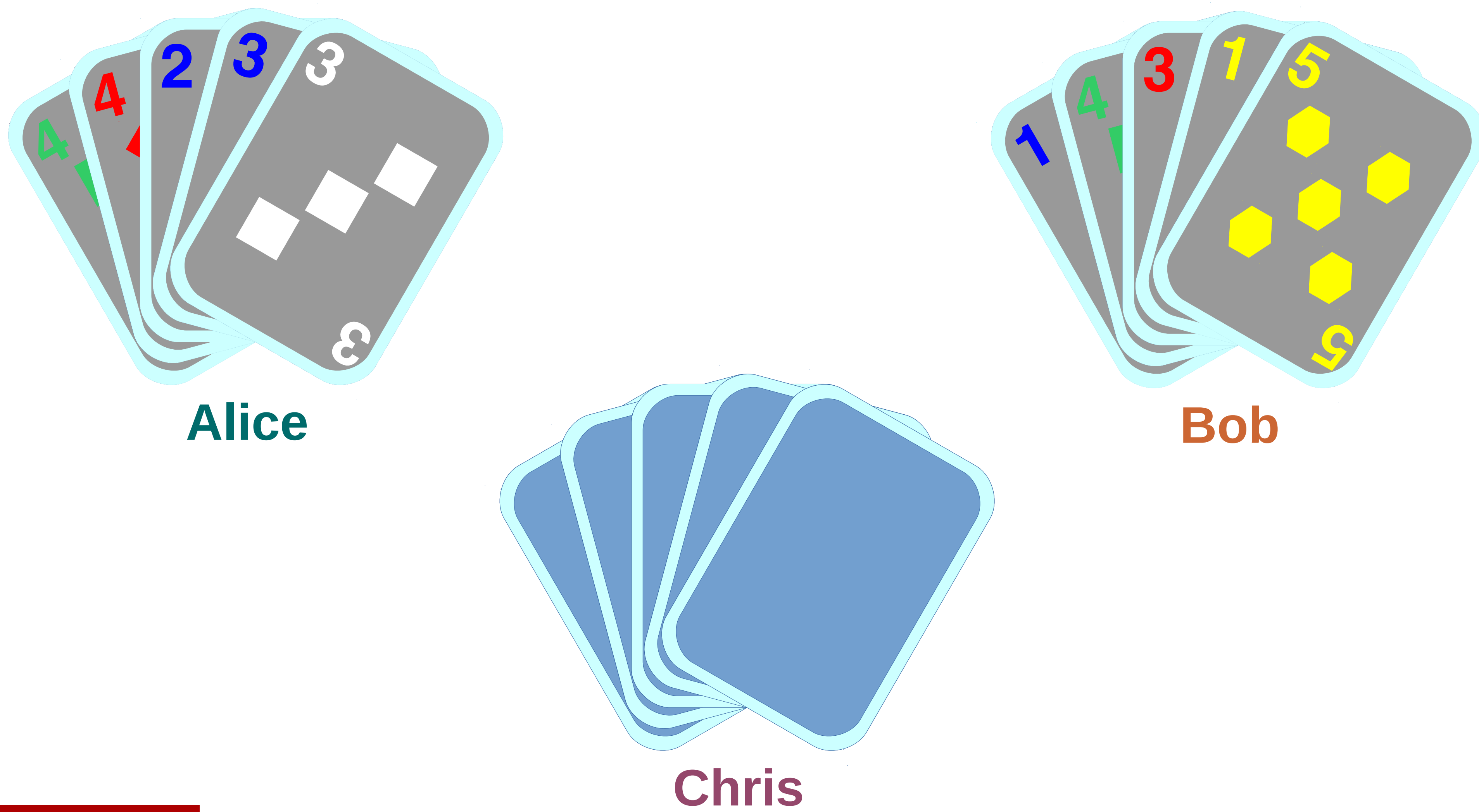
You have 5 reds.	You have 5 blues.	You have 5 greens.	You have 5 yellows.	You have 5 whites.
You have 4 reds.	You have 4 blues.	You have 4 greens.	You have 4 yellows.	You have 4 whites.
You have 3 reds.	You have 3 blues.	You have 3 greens.	You have 3 yellows.	You have 3 whites.
You have 2 reds.	You have 2 blues.	You have 2 greens.	You have 2 yellows.	You have 2 whites.
You have 1 reds.	You have 1 blues.	You have 1 greens.	You have 1 yellows.	You have 1 whites.
You have no reds.	You have no blues.	You have no greens.	You have no yellows.	You have no whites.
You have five 1's.	You have five 2's.	You have five 3's.	You have five 4's.	You have five 5's.
You have four 1's.	You have four 2's.	You have four 3's.	You have four 4's.	You have four 5's.
You have three 1's.	You have three 2's.	You have three 3's.	You have three 4's.	You have three 5's.
You have two 1's.	You have two 2's.	You have two 3's.	You have two 4's.	You have two 5's.
You have one 1.	You have one 2.	You have one 3.	You have one 4.	You have one 5.
You have no 1's.	You have no 2's.	You have no 3's.	You have no 4's.	You have no 5's.

Augmented by pointing, of course.

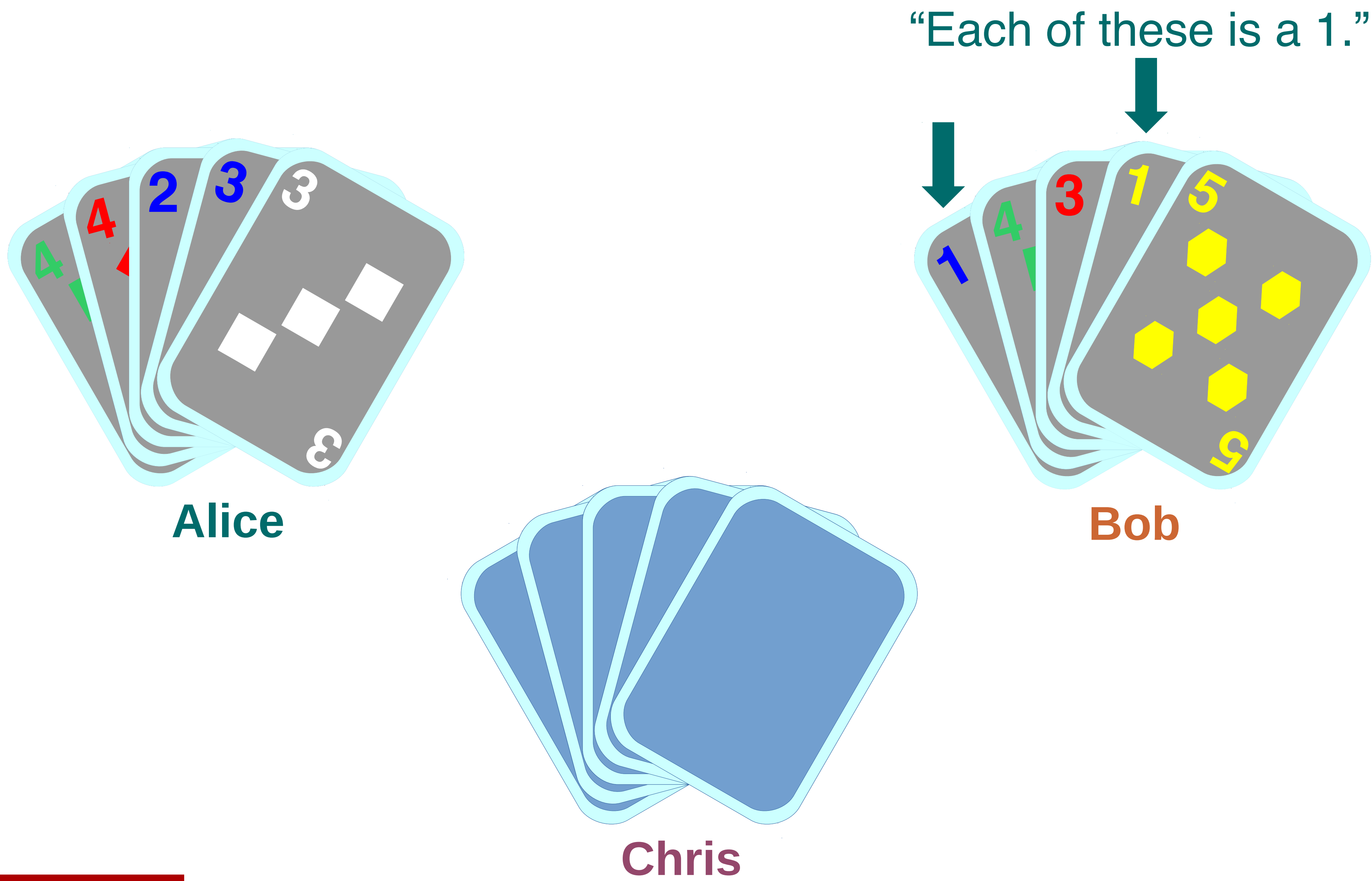
This is a small (and artificial) language.

ORACLE

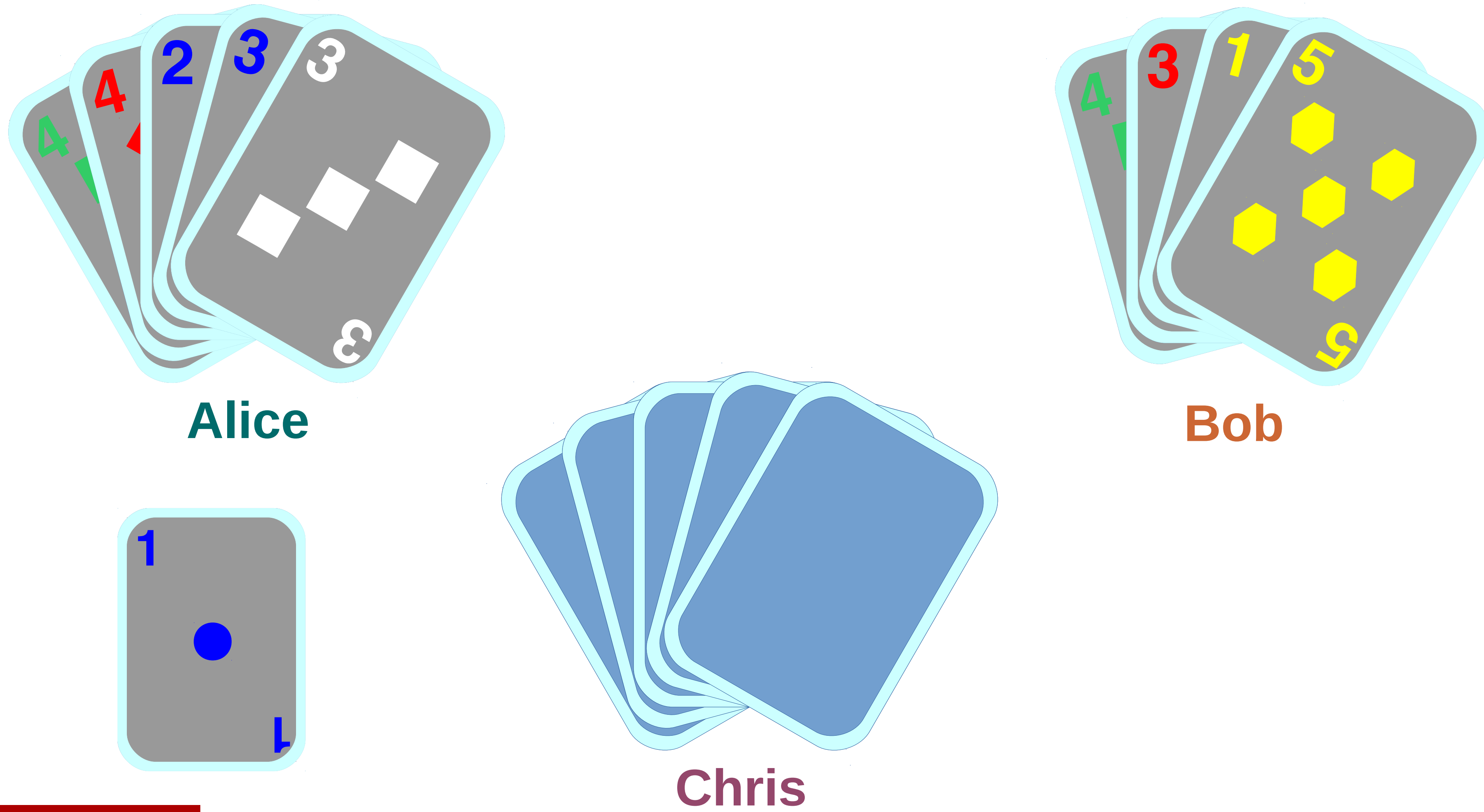
Hanabi: Sample Three-Player Game



Hanabi: Alice Gives Bob a Clue



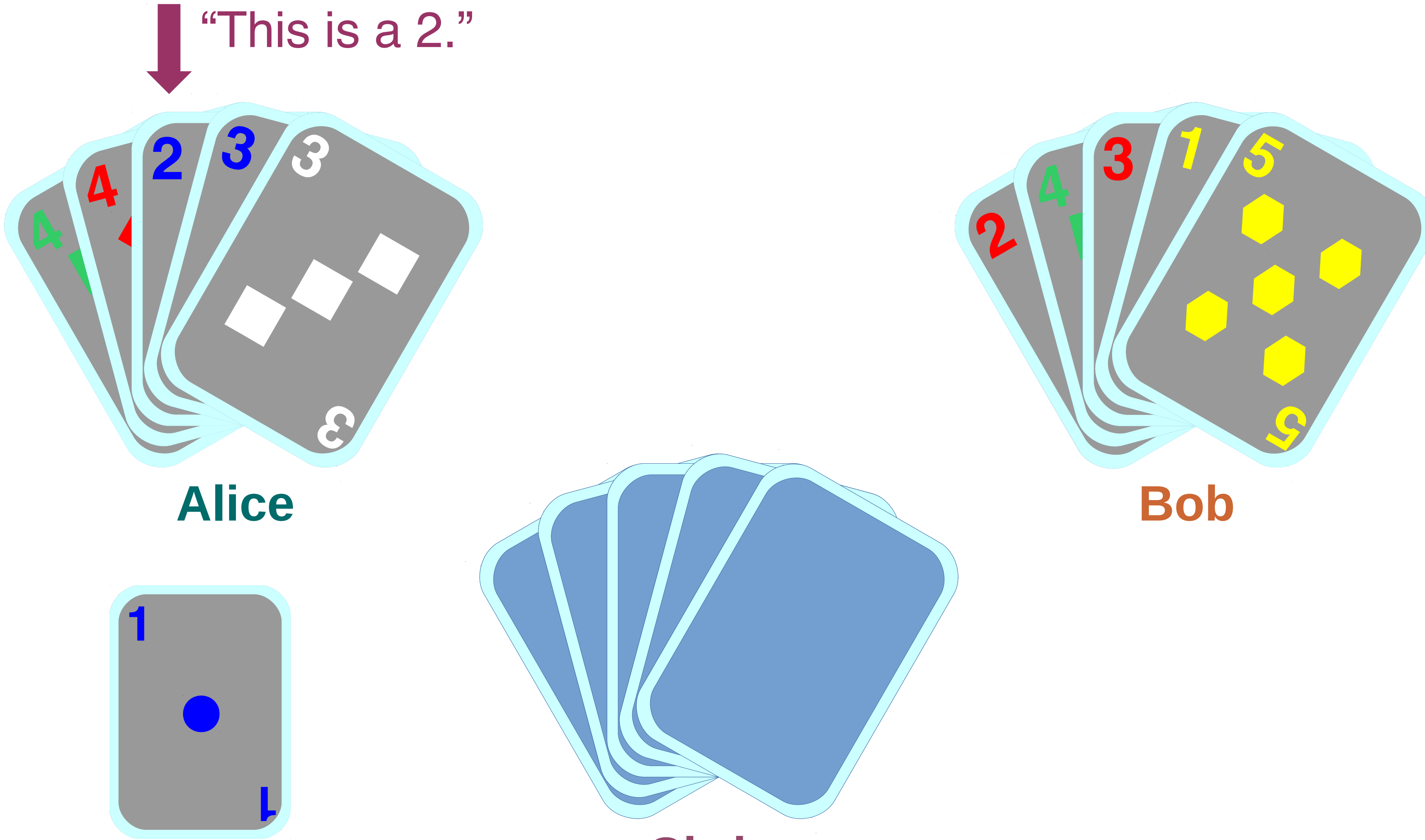
Hanabi: Bob Plays a Card . . .



Hanabi: . . . and Draws a New Card

Now what should Chris do?

Hanabi: Chris Gives Alice a Clue

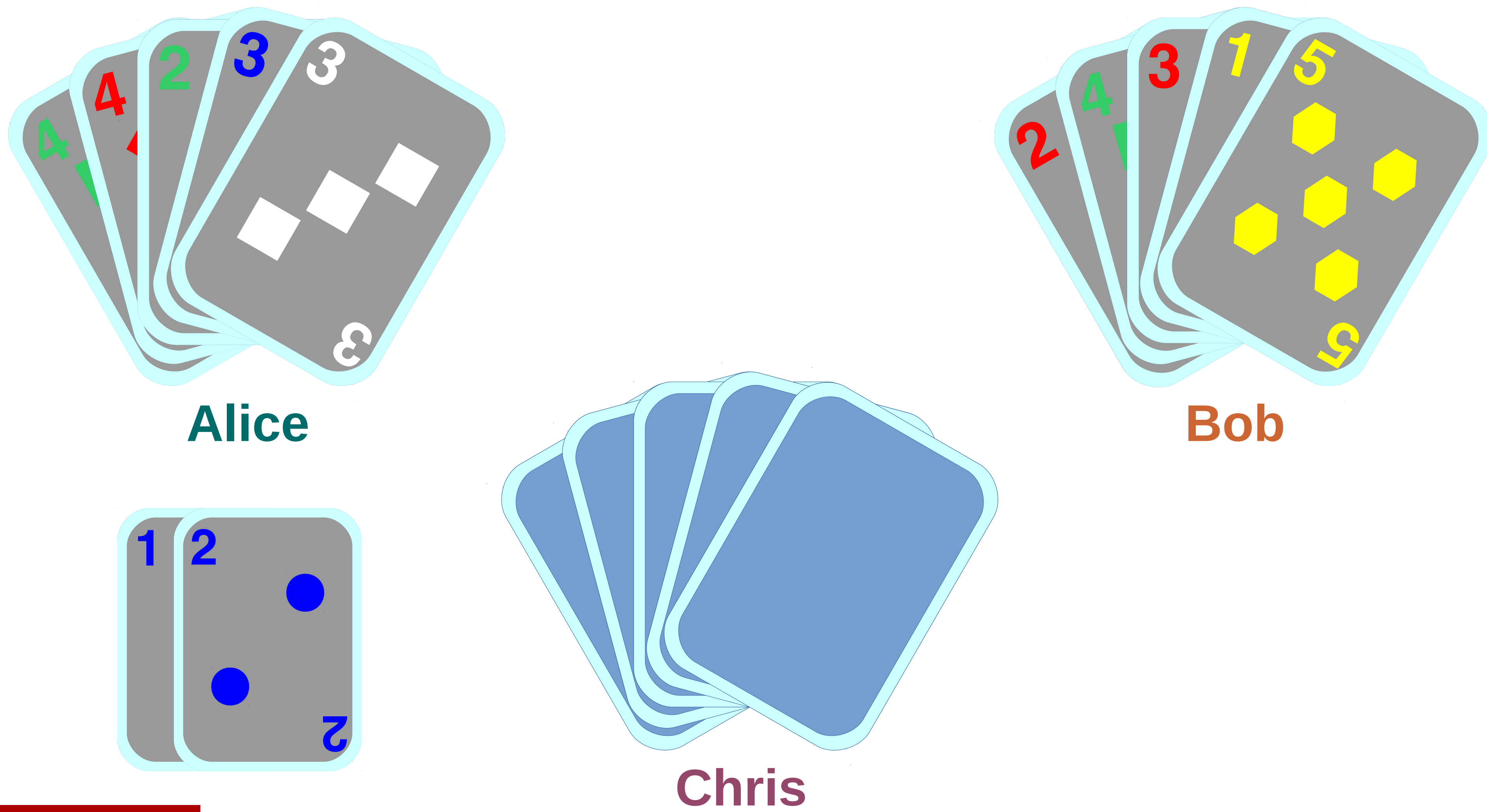


Now what should Alice do?

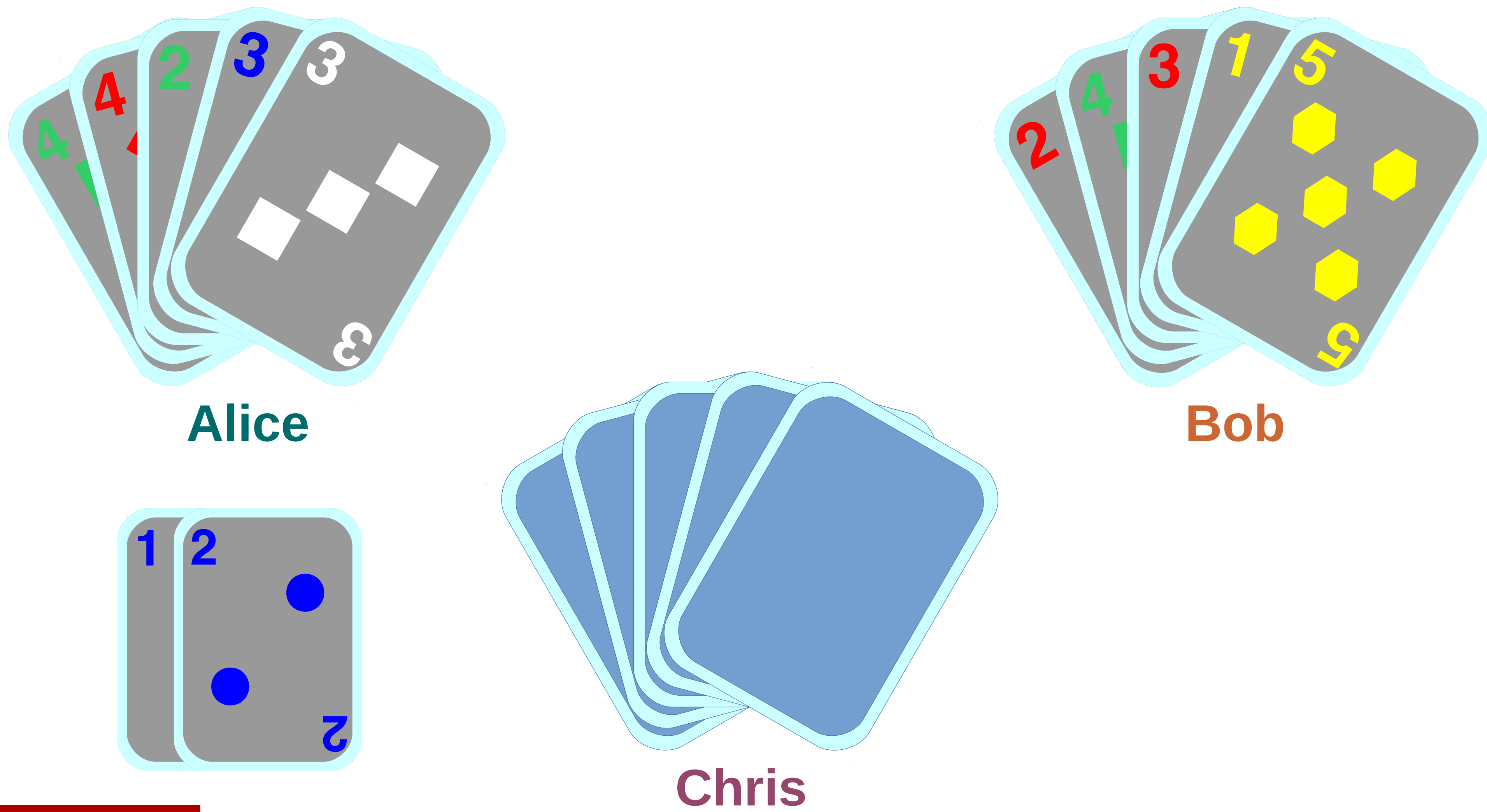
Hanabi: Alice Plays a Card . . .

ORACLE

Hanabi: . . . and Draws a New Card

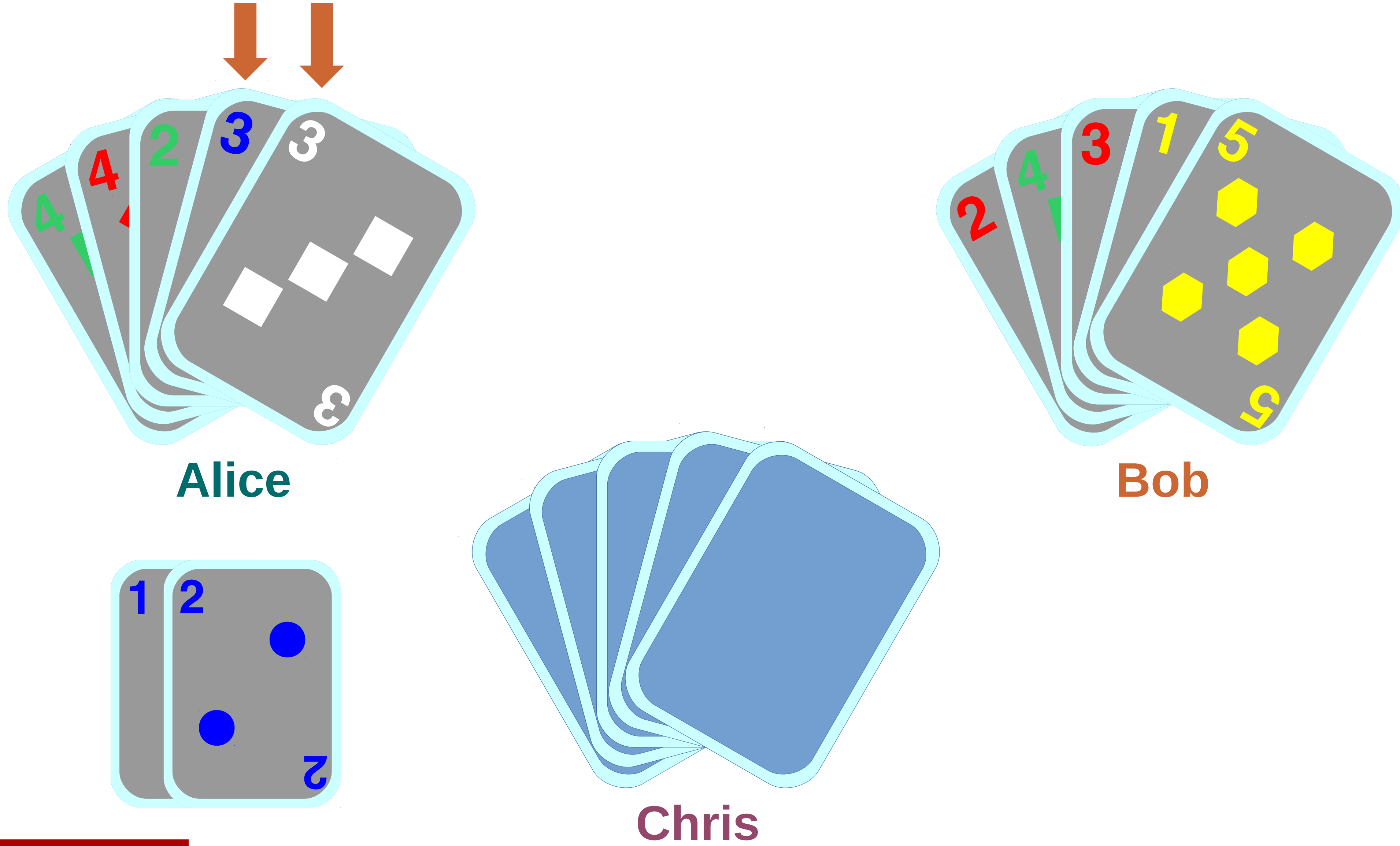


Hanabi: Should Bob Play His Yellow 1?

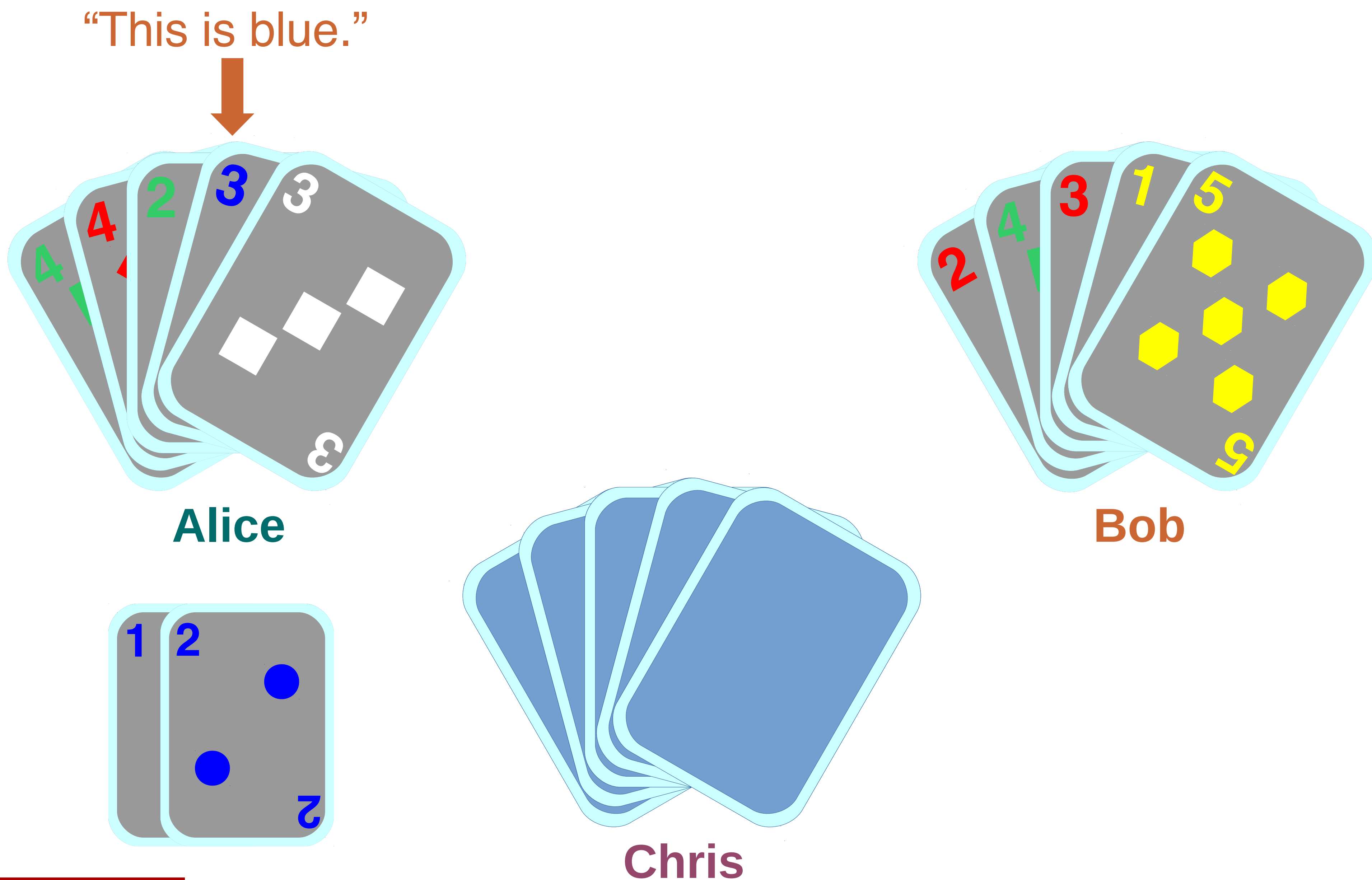


Hanabi: Bob Can Give Alice a Clue

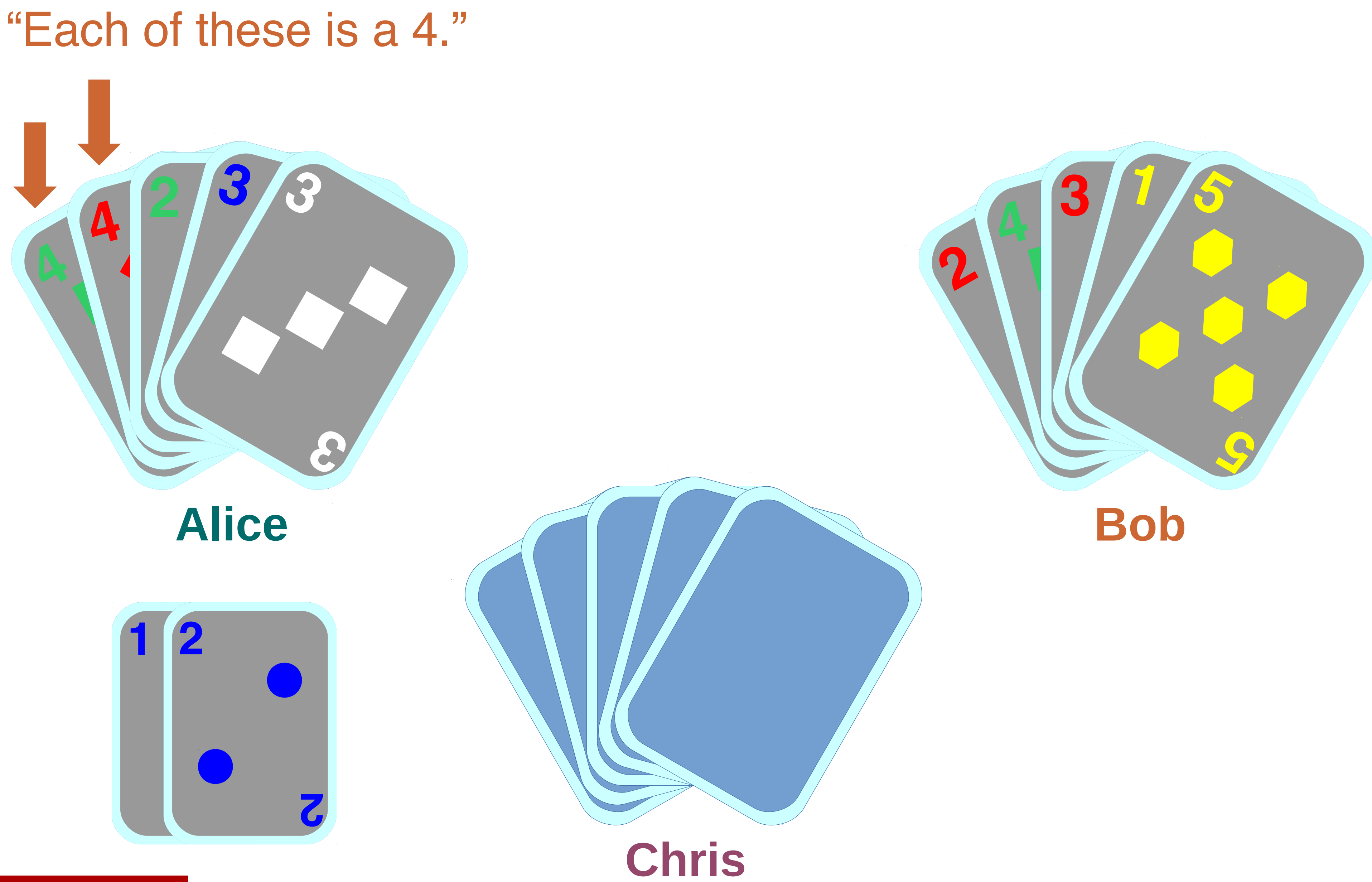
"Each of these is a 3."



Hanabi: Bob Can Give Alice an Actionable Clue



Hanabi: Bob Can Give Alice a Clue Clearly for Future Use (?)



Hanabi: One Could Give a Conventional Clue

One could invent elaborate conventions for Hanabi such as:

- The first time I give you a clue, if it's about color and points out exactly two cards, then it also implies that none of your cards is currently playable.
- If my clue is about color and points out exactly three cards, then it implies that both of the other cards are currently playable.

I haven't actually tried these, though I think they are plausible.

But a simple one I have used is:

- If I point out exactly *one* card and say anything other than "This is a 5," assume it is playable unless you can prove otherwise.

Such conventions are *beyond the rules of the game* and may be invented and adapted freely. Think of them as a kind of slang.

ORACLE

A Crazy Good Hanabi Strategy

For mathematical geeks: Hanabi is like a super hat-guessing puzzle.

With a sufficiently complicated convention, a single clue can give useful information to *every* other player simultaneously.

Christopher Cox, Jessica de Silva, Philip Deorsey, Franklin H. J. Kenter, Troy Retter, and Josh Tobin. How to make the perfect fireworks display: Two strategies for Hanabi. *Mathematics Magazine* 88, 5 (December 2015), 323–336. http://www.jstor.org/stable/10.4169/math.mag.88.5.323

ORACLE

Contract Bridge: The 38 Possible Utterances

7♣	7♦	7♥	7♠	7NT
6♣	6♦	6♥	6♠	6NT
5♣	5♦	5♥	5♠	5NT
4♣	4♦	4♥	4♠	4NT
3♣	3♦	3♥	3♠	3NT
2♣	2♦	2♥	2♠	2NT
1♣	1♦	1♥	1♠	1NT

Pass   Double   Redouble

ORACLE

Bidding in Contract Bridge

The bid “2♥” is a factual statement meaning, “If no one else bids higher, then my team will undertake to win at least 8 (that is, 6+2) tricks out of 13 with hearts (♥) as the trump suit.”

The bid “2♣” is a factual statement meaning, “If no one else bids higher, then my team will undertake to win at least 8 (that is, 6+2) tricks out of 13 with clubs (♣) as the trump suit.”

The bid “3NT” is a factual statement meaning, “If no one else bids higher, then my team will undertake to win at least 9 (that is, 6+3) tricks out of 13 with no trump suit.”

(In each case, there are scoring bonuses for success and penalties for failure.)

ORACLE

Bridge Bidding Conventions: What Do 2♥ and 2♣ Mean?

If you bid 1♥, partner bids 1NT, you bid 2♥: you really do want ♥ as trumps.
If you bid 1♣, partner bids 1NT, you bid 2♣: you really do want ♣ as trumps.

If partner opens 1NT and you respond 2♥, it means “Partner, please bid 2♠.”
If partner opens 1NT and you respond 2♣, it means “Partner, please:
  if you have at least 4 cards in ♥, bid 2♥; otherwise,
  if you have at least 4 cards in ♠, bid 2♠; otherwise, bid 2♦.”

If you open with 2♥, you have 6 cards in ♥ and a relatively weak hand.
If you open with 2♣, you have a very powerful hand (no promises about ♣).

Goal: communicate as much useful information as possible in as many situations as possible to win as many games as possible (there are tradeoffs).

ORACLE

“It’s raining.”

In a conversational context,
the interpretation of a proposition
can depend on not only
beliefs about whether it is *true*
but also beliefs about
context and *relevance* and *intention*.

This can make conversation *more efficient*.

This Talk Is an Essay (I Didn't Know Where It Would Go)

I started out wanting to tell things to a compiler (or IDE).

- Specifically, I want to tell a compiler far more than types.
- I thought the conclusion would be that compilers need theorem provers.

That's not a bad goal. But I have ended up wanting much more:

- I want a conversational partner that will track what I am doing.
- I want it to react to context and intention.
- I want it to give me relevant information.

This is much harder than "Just the facts, Ma'am."

ORACLE

"Let's change the type of `x`

from `short` to `long`."

“These arrays should be kept sorted.”

“Let’s try the same set of
loop-interchange transformations
that we used last week
on that other algorithm.”

Questions?

Comments?