CONVERGENT EVOLUTION

Arriving at C++'s Concepts from a different path.

THERE'S NO GREAT INSIGHT HERE

- What I'm about to present is well-known information…
  - …if you've read a lot of papers on the subject.
  - If you haven't read any papers, you might think that concepts are just a way to simplify template error messages.
- But the ideas here are *cool*!
  - I want to disseminate them to show you how cool they are.
- May help you reason about your programs in a new way.
- Also demonstrates that sometimes there really is only a single right answer when designing things.

PROGRAMS ARE PROOFS

- Called the "Curry-Howard correspondence".
  - Types are propositions.
  - An object with that type proves the proposition that that type represents.
- Not 100% true in any real life programming system, *especially* not C++.
  - Evaluation scheme, side effects, cosmic rays, etc.
- But it's fun to think that way sometimes.
  - Productive? I don't know.
- **This idea ties all programming languages together.**

WHAT DOES IT MEAN?

- Functions are implications (IF):
  - Haskell: A -> B
  - B (*)(A)
  - std::function<B(A)>, I guess?
- Aggregates are conjunctions (AND):
  - struct { A a; B b; }
  - std::tuple<A, B>
  - *Many* ways in a lot of languages.
- Sum types are disjunctions (OR):
  - union {A a; B b}
  - Haskell: data Foo = FooA A | FooB B
- Parametric polymorphism is "for all" (∀), also known as "universal types":
  - Haskell: id :: forall a. a -> a
  - C++: template<typename A> A id(A);
- Existential types (∃) are complicated.
  - TaPL (Pierce) says they're pairs of values and types.
  - ML modules?
  - Encodable with universals anyway.

WHAT IS A CONSTRAINT?

- Say you want to sort a set. You have to prove that the elements of that set have a total order.
- void qsort(void *base, size_t num, size_t size, int (*compar)(const void *, const void *))
  - "compar" here is the *proof* that the order exists.
  - It is a *constraint* on the elements of the set.
  - Yes, this is a *very* rough approximation.
- **A constraint is an *extra parameter*.**
- They're just lemmas in your proof.

SO… WHY CONCEPTS/TYPE CLASSES?

- Convenience!
- If it's an *established fact* that objects of type T have total order, then the sorting function shouldn't require me to explicitly supply that proof every time I want to sort a set of T objects.
- You're telling a function to JFGI.
  - Just f***ing get it from the environment!
  - Typeclasses in Haskell and Scala are actually *implicit parameters* indexed by *types* (yes, typeclasses are types too).
- Concepts and typeclasses allow you to state facts about types in a single place and let those facts be known everywhere, just by mentioning the constraint.

DRINK THE KOOL-AID

Haskell

```
sort :: Ord a => [a] -> [a]

class Eq a => Ord a where
        (<=) :: a -> a -> Bool

class Eq a where
        (==) :: a -> a -> Bool
```

C++

```
template<typename T> requires Ord<T>
list<T> sort(list<T>);

template<typename T>
concept bool Ord = requires (T a, T b)
        { {a <= b} -> bool; ... }

template<typename T>
concept bool Eq = requires (T a, T b)
        { {a == b} -> bool; ... }
```

FURTHER READING

- Oliveira, Moors, Odersky: Type Classes as Objects and Implicits. 2010.
- Bernardy, Jansson, Zalewsky, Schupp: Generic Programming with C++ Concepts and Haskell Type Classes – A Comparison. 2010.