Jumbo ML

Language Support for CS2

Norman Ramsey

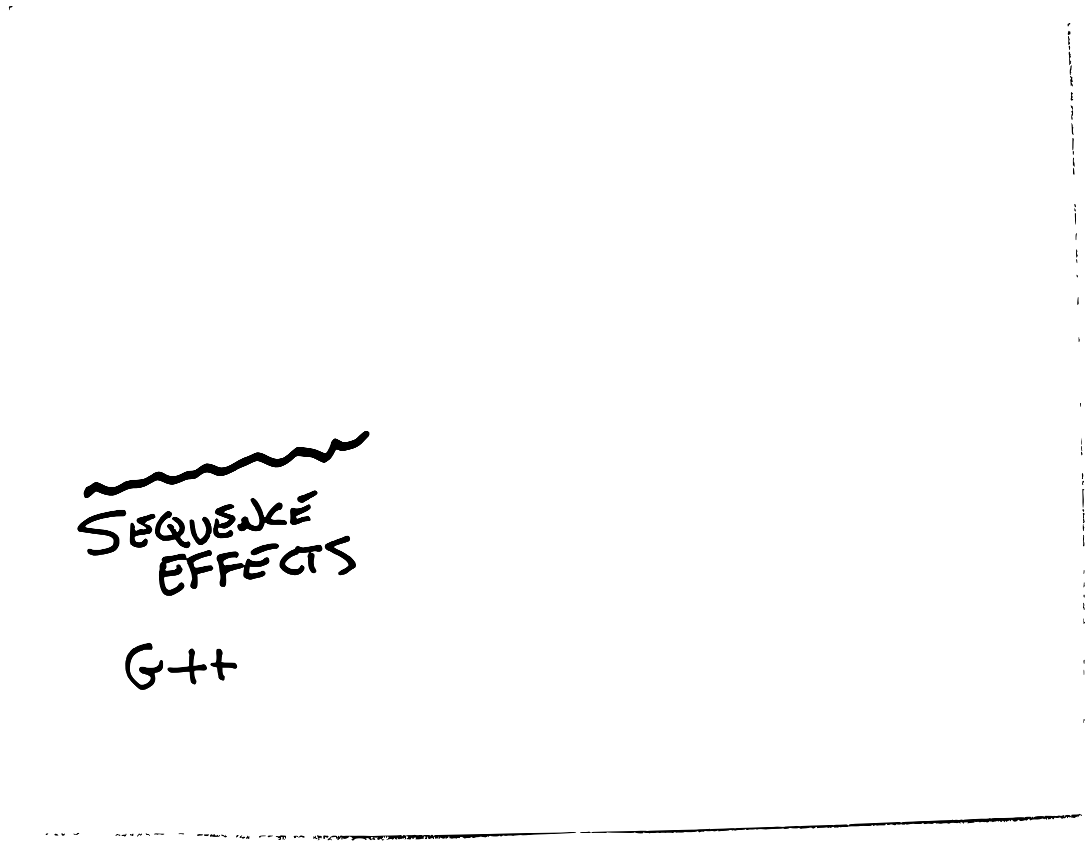
Tufts University

Jumbo ML addresses two problems

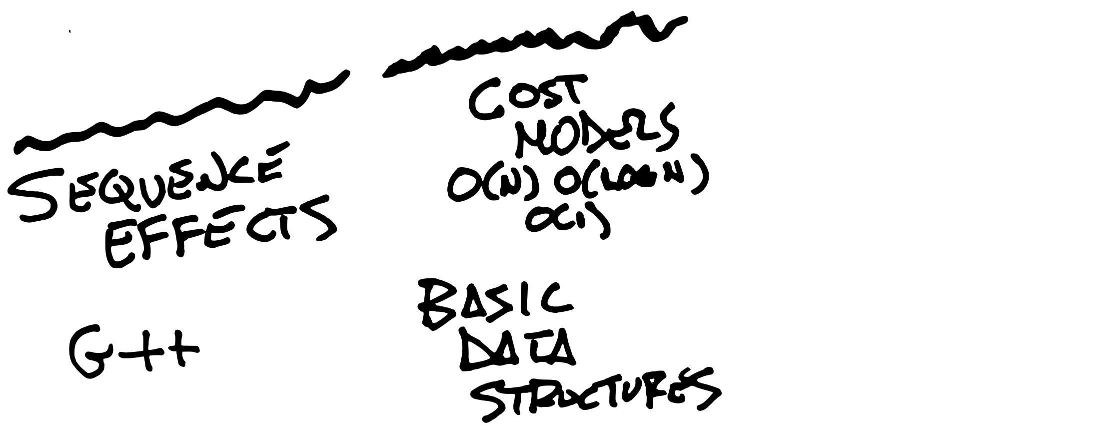
Support for curricular revision

Sandbox for beginning research students

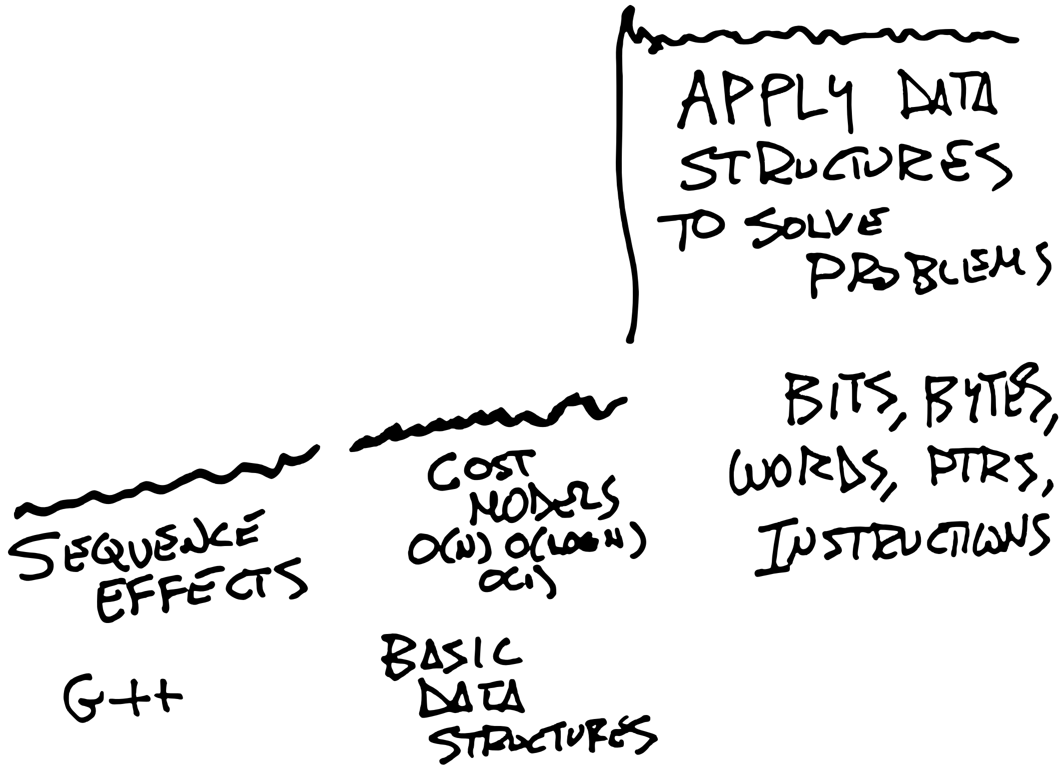
Status quo ante contains a cliff



Status quo ante contains a cliff



Status quo ante contains a cliff



Cliff shows in projects

CS3: Black-edge removal:  => 

Cliff shows in projects

CS3: Black-edge removal:  => 

JPEG compression/decompression

Attack cliff with baby steps

CS2: ***Song Search***

CS3: Black-edge removal:  **=>** 

JPEG compression/decompression

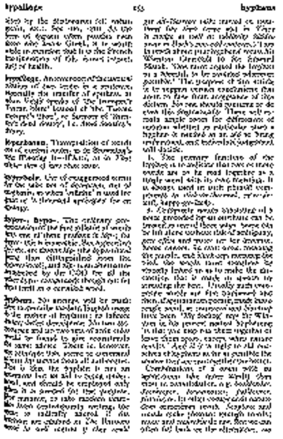
Attack cliff with baby steps

CS1: What language is this web page? (*Trigrams*)

CS2: *Song Search*

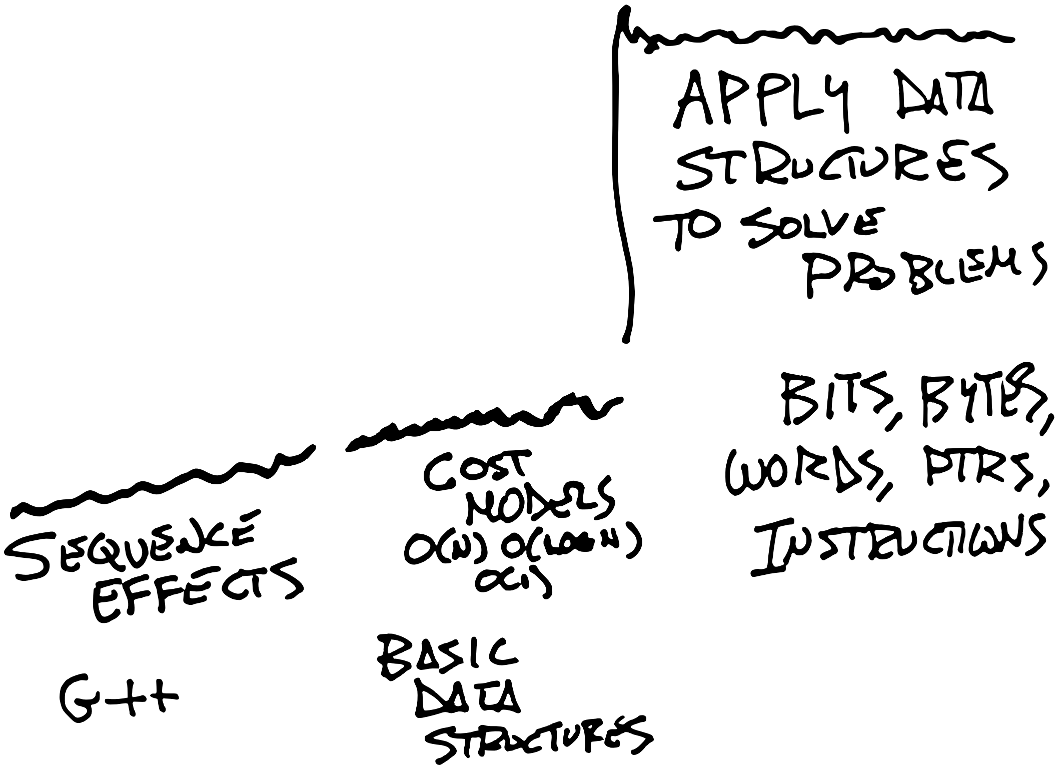
CS3: Black-edge removal:

=>

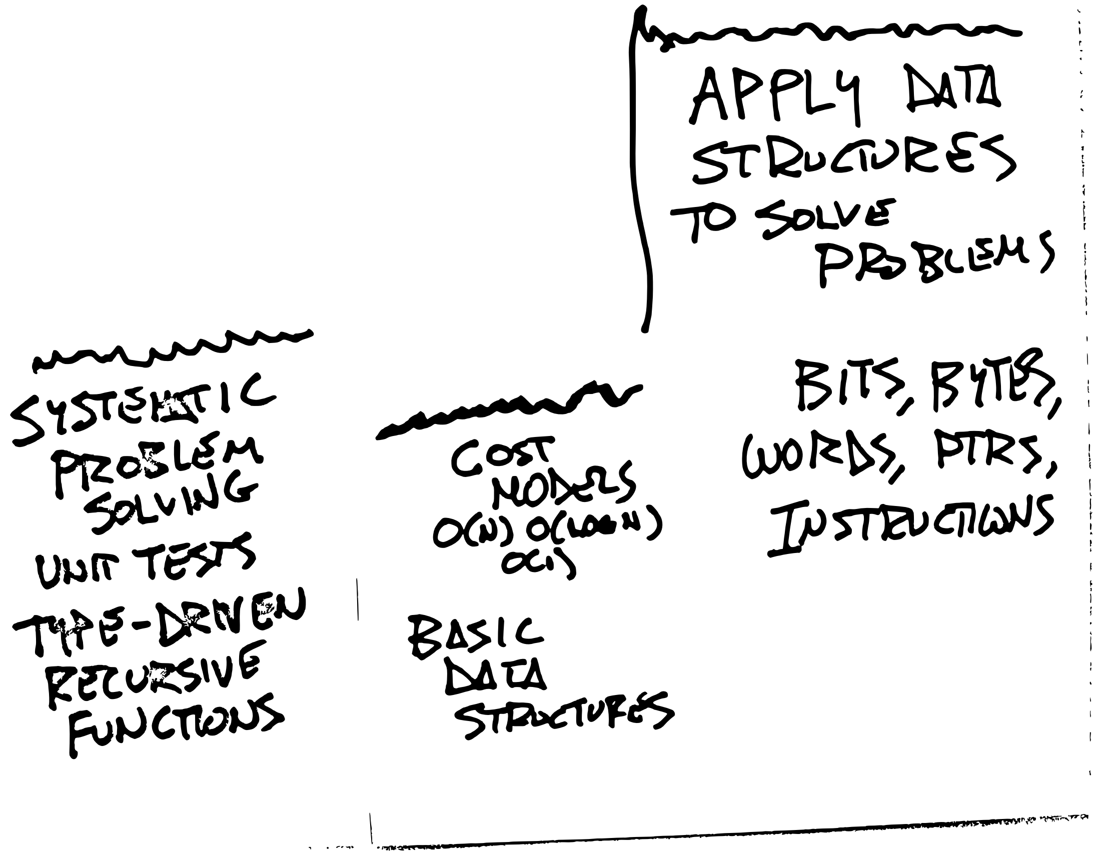


JPEG compression/decompression

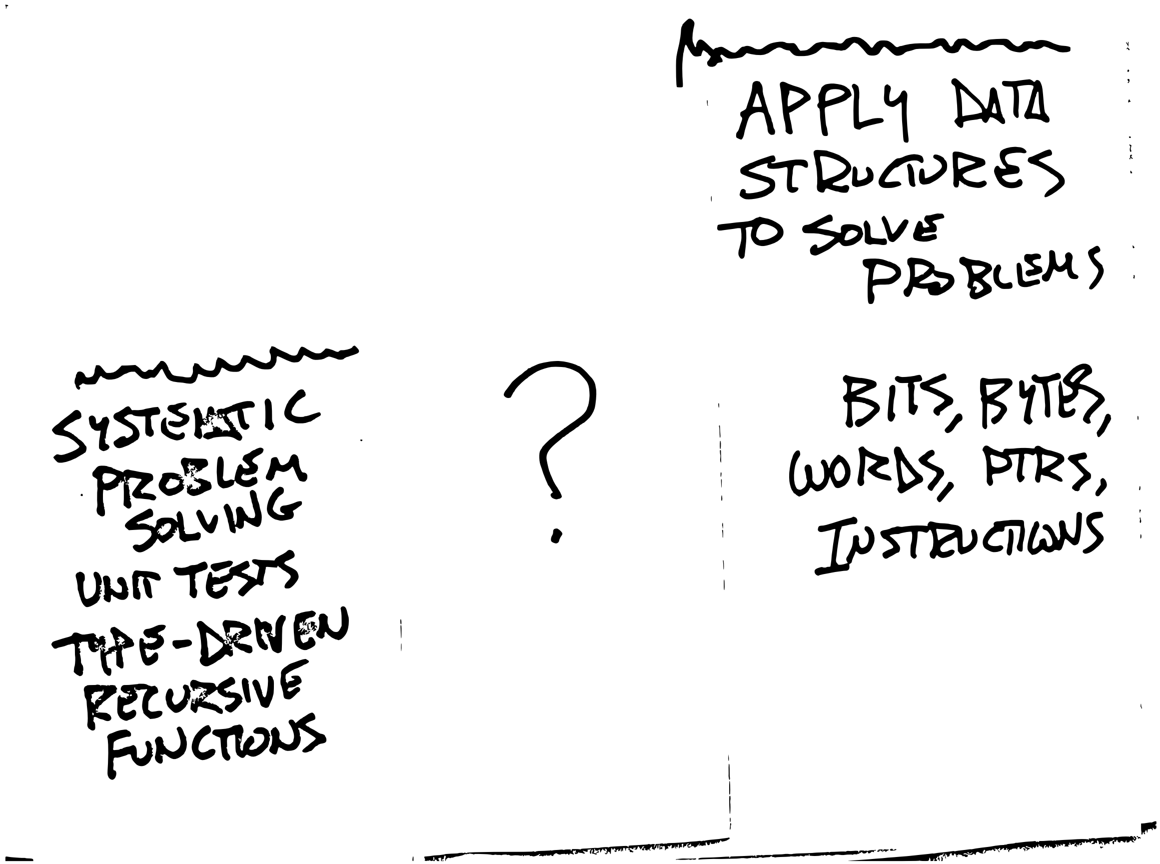
The cliff is still there



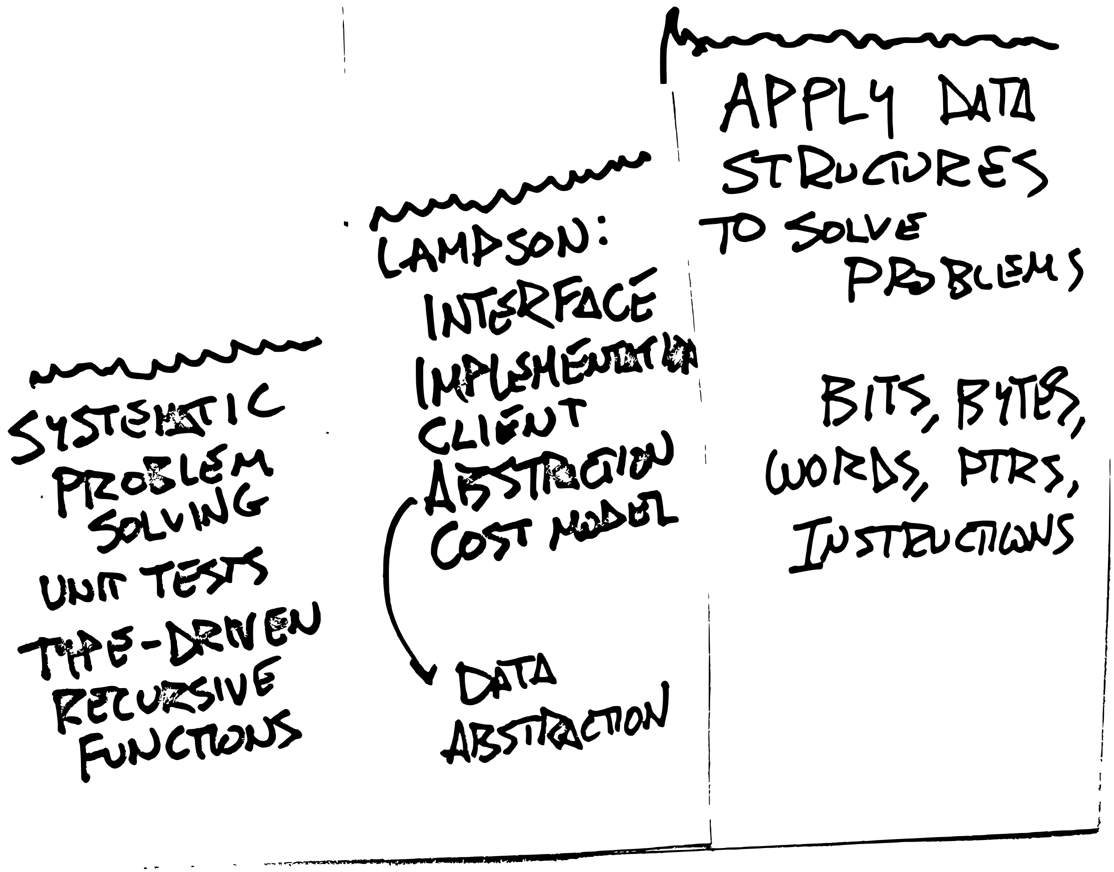
To reduce cliff, shift learning goals



To reduce cliff, shift learning goals



To reduce cliff, shift learning goals



Top learning goals for CS2

Dip toes into Butler Lampson’s pool

A design is a collection of interfaces

Data structures not on stone tablets

Abstraction + cost model = data structure

Supported by what language?

Learning goals drive language needs

Separately compiled interfaces (w/static types)

Code against interfaces that aren't yet implemented

Data abstraction

Perspicuous cost model

Bridge teaching languages $\longrightarrow$ industrial languages

Standard ML isn’t simple enough

Problems already encountered in our teaching:

- **Core language lacks type signatures**
- **Concrete syntax has traps and pitfalls**
- **Allocation costs aren’t so obvious**
- **In polymorphic types, `forall` is implicit**
- **Value restriction baffles students**
- **Fibration (`where type t = t`) *really* baffles ’em**

Simplicity looks like *SL

Beginning Student Language:

- **Type is *atomic data*, *sum*, or *named product***
- **Syntax is**

$$
\begin{aligned}
\textit{exp} \Rightarrow\ & x \\
\mid\ & v \\
\mid\ & (x\ \textit{exp}_1 \ldots \textit{exp}_n) \\
\mid\ & (\texttt{cond}\ \{(\textit{exp}\ \textit{exp})\}\ [(\texttt{else}\ \textit{exp})]) \\
\mid\ & (\texttt{and}\ \textit{exp}\ \textit{exp}) \mid (\texttt{or}\ \textit{exp}\ \textit{exp}) \\
\textit{def} \Rightarrow\ & (\texttt{define}\ (x\ x_1 \ldots x_n)\ \textit{exp}) \\
\mid\ & (\texttt{define}\ x\ \textit{exp}) \\
\mid\ & (\texttt{define-struct}\ x\ (x_1 \ldots x_n)) \\
\mid\ & \textit{unit-test}
\end{aligned}
$$

Simplicity looks like *SL

Intermediate Student Language (with lambda) adds

- **First-class functions**
- **Let-bindings**

Add *arrow types*

Add these terms:

$$\begin{array}{rcl} \textit{exp} & \Rightarrow & \texttt{(}\textit{exp}\ \textit{exp}_1 \ldots \textit{exp}_n\texttt{)} \\ & | & \texttt{(lambda (}x_1 \ldots x_n\texttt{)}\ \textit{exp}\texttt{)} \\ & | & \texttt{(local (}\{\textit{def}\}\texttt{)}\ \textit{exp}\texttt{)} \end{array}$$

“Systems thinking” needs more complexity

Consensus irreducible complexity:

terms	types
definitions	declarations
modules	module types

“Systems thinking” needs more complexity

Consensus irreducible complexity:

terms	types
definitions	declarations
modules	module types

Hidden irreducible complexity?

Compilation unit?

Component?

Program?

Let’s talk

Facing many decisions:

- **Definitions: letrec, let-star, or “mostly letrec”?**
- **What should sum types look like?**
  What about pattern matching?
- **Include exceptions? What kind?**
- **How to reflect purity in types?**
- **Components: modules, mixins, units?**

Sequencing definitions matters

Let-star (ML), letrec (Haskell), “mostly letrec” (Scheme):

	ML	Haskell	Scheme
Teach def before use	✓	☹	✓
Easy type inference	✓	✓	?
Easy core type signatures	☹	✓	?
Easy mutual recursion	☹	✓	✓
Simple theory	✓	✓	☹

This week: understand Scheme model better

Pattern matching should be restricted

Decided:

- **No anonymous tuples or flex records (want constructor function for allocation)**
- **Pattern matching only in `case`**
- **No pattern matching on products?**

Pattern matching should be restricted

Decided:

- No anonymous tuples or flex records (want constructor function for allocation)
- Pattern matching only in `case`
- No pattern matching on products?

Known problems with algebraic data types:

- Pattern matching militates against abstraction
- Bundle sum tagging with recursion, generativity

Wanted: simpler sums

Goals, ideas:

- “Mostly letrec:” all types (mutually) recursive
- Unbundled generativity (Modula-3 `BRANDED`)
- Keep checks for exhaustive, redundant patterns?

What should sum elimination look like?

- ML-like with a binding construct?
- CLU-like with selector functions?

Exceptions come in many flavors

From least flexible to most flexible:

- Checked run-time error (C `abort`)
- Checked RTE with recovery (Erlang)
- Alternate return for abstract data type (CLU)
- “Oh dear, I can’t go on” (Haskell)
- Control operator, reflected in type (Java)
- Control operator, anything goes (ML)

CLU has a teaching story. Do others?

Effects should be controlled

Purity claimed as major benefit of Haskell (PJ'03)

BSL, ISL are pure

Purity can eliminate the value restriction

Effects should be controlled

Purity claimed as major benefit of Haskell (PJ'03)

BSL, ISL are pure

Purity can eliminate the value restriction

Jumbo ML needs I/O, mutable data

How? Basic Jumbo ML won't have type classes

- Should it have an IO monad? With `do` notation?
- Would a type-and-effect system be better?

Modules/Components: Too many choices

ML functors: The devil I know

Typed units (Owens and Flatt 2006): Attractive, complex

Backpack (Kilpatrick *et al* 2014): Simpler, less expressive

Many module calculi (Dreyer 2003–2014): Core calculi

This week: get advice, hear war stories

I’m eager to talk

Much more at

`http://www.cs.tufts.edu/~nr/pubs/jml-aug14.pdf`