Embedding languages

Chung-chieh Shan

There’s treasure everywhere:
a Calvin and Hobbes collection by Bill Watterson

Oatmeal pancakes

Soak 3/4 cup oats in 3/4 cup water.

In a separate bowl, mix:

- 1 cup whole wheat flour
- 2 tablespoons flax seed meal
- 3 teaspoons baking [powder]
- 2 tablespoons sugar (evaporated cane sugar or whatever, please no bone sugar!)
- Cinnamon, allspice, fresh grated nutmeg “to taste”

Add the soaked oats with water to the flour mixture.

Add soymilk to make a good thick mixture.

Cook in a medium hot skillet with light olive oil . . .

(Emily Thurston)

Interpreting recipes

Make it.
Reserve kitchen equipment.
Typeset it.
Is it vegetarian?

How long does it take to make?
How many calories does it have?
How much does it cost?
How much cinnamon to add?

Interpreting recipes

Make it.

Reserve kitchen equipment.

Typeset it.

Is it vegetarian?

How long does it take to make?

How many calories does it have?

How much does it cost?

How much cinnamon to add?

```
pancakes
 = cook (mix [soak (3/4) cup (measure (3/4) cup oats),
              measure 1 cup whole_wheat_flour,
              measure 2 tablespoon flax_seed_meal,
              measure 3 teaspoon baking_powder,
              measure 2 tablespoon sugar,
              ...],
         ...)
```

Representing knowledge as programs

Some examples:

- recipes
- contracts (stock options)
- decision processes (games)
- grammars (`printf` formats, regular expressions)
- media (music, animation)
- user interfaces (layout, validation)
- natural language

Representing knowledge as programs

Some examples:

- recipes
- contracts (stock options)
- decision processes (games)
- grammars (`printf` formats, regular expressions)
- media (music, animation)
- user interfaces (layout, validation)
- natural language

Whether procedural or declarative—
What is a program?

- executable
- composable
- expressive
- intuitive

Outline

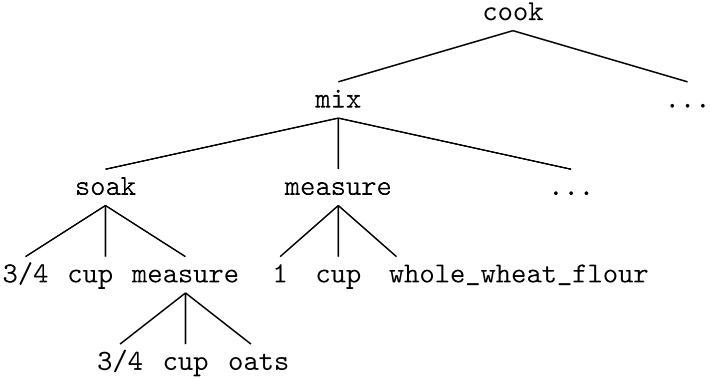
- **Representing knowledge as programs**
  - Recursive syntactic structure
  - Multiple semantic interpretations
  - Binding and procedural abstraction
  - Types

- Embedding languages
  - Tagging overhead
  - Common subexpressions
  - Embedding interpreters

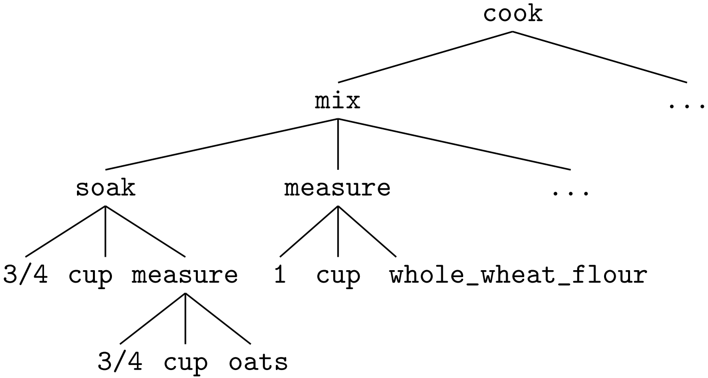
- Preserving types and binding
  - Finally tagless
  - Closing the stage

?

Recursive syntactic structure



Recursive syntactic structure



$$E ::= \texttt{mix}\,[E, \ldots\,] \mid \texttt{soak}\; n\; U\; E \mid \texttt{measure}\; n\; U\; E \mid \texttt{oats} \mid \ldots$$

$$U ::= \texttt{cup} \mid \texttt{tablespoon} \mid \texttt{teaspoon} \mid \ldots$$

Multiple semantic interpretations

Many back-ends: action, text, nutrition, cost, time, policy, . . .

Multiple semantic interpretations

Many back-ends: action, text, nutrition, cost, time, policy, ...
We prefer a bottom-up (compositional) interpreter.

$$
\begin{aligned}
[\![\texttt{oats}]\!] &= 300\,\mathrm{kcal} : 1\,\mathrm{cup} : 80\,\mathrm{gram} \\
[\![\texttt{water}]\!] &= 0\,\mathrm{kcal} : 1\,\mathrm{cup} : 230\,\mathrm{gram} \\
[\![\texttt{measure}\ n\ \texttt{cup}\ E]\!] &= \frac{xn}{y}\,\mathrm{kcal} : n\,\mathrm{cup} : \frac{zn}{y}\,\mathrm{gram} \\
&\quad \text{where } [\![E]\!] = x\,\mathrm{kcal} : y\,\mathrm{cup} : z\,\mathrm{gram} \\
[\![\texttt{mix [}E_1, \ldots, E_n\texttt{]}]\!] &= \sum_{i=1}^{n} [\![E_i]\!]
\end{aligned}
$$

(Ignoring fine points about chemistry and ratios.)

Binding

```
seasoning = mix [measure 1 teaspoon cinnamon,
                 measure 1 teaspoon allspice,
                 measure 1 teaspoon (grate nutmeg)]
```

Bound variables!

Binding and procedural abstraction

```
seasoning = mix [measure 1 teaspoon cinnamon,
                 measure 1 teaspoon allspice,
                 measure 1 teaspoon (grate nutmeg)]

soak n u x = wait (mix [x, measure n u water])
```

Bound variables!
Functions!

Binding and procedural abstraction

```
seasoning = mix (map (measure 1 teaspoon)
                     [cinnamon, allspice,
                      grate nutmeg])

soak n u x = wait (mix [x, measure n u water])

map f []        = []
map f (x :: xs) = f x :: map f xs
```

Bound variables!
Functions!
Callback (higher-order) functions!   “one teaspoon each of . . . ”

Types

Classify terms more finely.

$$T ::= \text{food} \mid \text{number} \mid \text{unit} \mid T\ \text{list} \mid T \to T$$

Types

Classify terms more finely.

$$T ::= \mathrm{food} \mid \mathrm{number} \mid \mathrm{unit} \mid T \text{ list} \mid T \to T$$

$$\frac{}{\texttt{water} : \mathrm{food}} \qquad \frac{\begin{array}{c}[x : T_1] \\ \vdots \\ E : T_2\end{array}}{\lambda x.\, E : T_1 \to T_2} \qquad \frac{E_1 : T_1 \to T_2 \quad E_2 : T_1}{E_1(E_2) : T_2}$$

Types

Classify terms more finely.

$$T ::= \mathrm{food} \mid \mathrm{number} \mid \mathrm{unit} \mid T\ \mathrm{list} \mid T \to T$$

$$\frac{}{\texttt{water} : \mathrm{food}} \qquad \frac{\begin{array}{c}[x : T_1] \\ \vdots \\ E : T_2\end{array}}{\lambda x.\, E : T_1 \to T_2} \qquad \frac{E_1 : T_1 \to T_2 \quad E_2 : T_1}{E_1(E_2) : T_2}$$

$$\texttt{seasoning} : \mathrm{food} \qquad \texttt{soak} : \mathrm{number} \to \mathrm{unit} \to \mathrm{food} \to \mathrm{food}$$

Types

Classify terms more finely.

$$T ::= \mathrm{food} \mid \mathrm{number} \mid \mathrm{unit} \mid T\ \mathrm{list} \mid T \to T$$

$$\frac{}{\mathtt{water} : \mathrm{food}} \qquad \frac{\begin{array}{c}[x : T_1] \\ \vdots \\ E : T_2\end{array}}{\lambda x.\, E : T_1 \to T_2} \qquad \frac{E_1 : T_1 \to T_2 \quad E_2 : T_1}{E_1(E_2) : T_2}$$

$$\mathtt{seasoning} : \mathrm{food} \qquad \mathtt{soak} : \mathrm{number} \to \mathrm{unit} \to \mathrm{food} \to \mathrm{food}$$

Further distinctions: mass (oats) vs count (pancakes),
carnivore (bone sugar) vs vegetarian (cane sugar), . . .
Static safety guarantees.

Types

Classify terms more finely.

$$T ::= \mathrm{food} \mid \mathrm{number} \mid \mathrm{unit} \mid T\ \mathrm{list} \mid T \to T$$

$$\frac{}{\texttt{water} : \mathrm{food}} \qquad \frac{\begin{array}{c}[x : T_1] \\ \vdots \\ E : T_2\end{array}}{\lambda x.\, E : T_1 \to T_2} \qquad \frac{E_1 : T_1 \to T_2 \quad E_2 : T_1}{E_1(E_2) : T_2}$$

$$\texttt{seasoning} : \mathrm{food} \qquad \texttt{soak} : \mathrm{number} \to \mathrm{unit} \to \mathrm{food} \to \mathrm{food}$$

Further distinctions: mass (oats) vs count (pancakes), carnivore (bone sugar) vs vegetarian (cane sugar), …
Static safety guarantees.

Like terms, types also have recursive syntax, multiple semantics, binding, procedures.

Outline

- Representing knowledge as programs
  - Recursive syntactic structure
  - Multiple semantic interpretations
  - Binding and procedural abstraction
  - Types

- ► **Embedding languages**
  - Tagging overhead
  - Common subexpressions
  - Embedding interpreters

- Preserving types and binding
  - Finally tagless
  - Closing the stage

Domain-specific languages

Some examples:

- recipes
- contracts (stock options)
- decision processes (games)
- grammars (`printf` formats, regular expressions)
- media (music, animation)
- user interfaces (layout, validation)
- natural language
- …

Better together

Embedding languages in each other:

- downloading and parsing recipes
- generating and running shaders and SQL
- "to taste"
- *theory of mind*:
  Object values ≈ what the modeled agent knows
  Object terms ≈ what the modeling agent believes
- *mixed quotation*:
  Bush also said his administration would "achieve our objectives" in Iraq. (New York Times, November 4, 2004)
  Logic and Engineering of Natural Language Semantics 2007.
  Amsterdam Colloquium 2007.

Better together

Embedding languages in each other:

- downloading and parsing recipes
- generating and running shaders and SQL
- "to taste"
- *theory of mind*:
  Object values ≈ what the modeled agent knows
  Object terms ≈ what the modeling agent believes
- *mixed quotation*:
  Bush also said his administration would "achieve our objectives" in Iraq. (New York Times, November 4, 2004)
  Logic and Engineering of Natural Language Semantics 2007.
  Amsterdam Colloquium 2007.

Nested containers—but with recursive syntax,
multiple semantics, binding, procedures, types.

Programs as data

The central question:

How to represent object programs in the metalanguage?

Desiderata:

- Multiple interpretations
- Preserve types and binding
- Preserve sharing
- Embed interpreters

Programs as data

```
String pancakes = "cook (mix [...], ...)";
double kcal = Nutrition.interpret(pancakes);
```

(cf. POSIX `regcomp`)

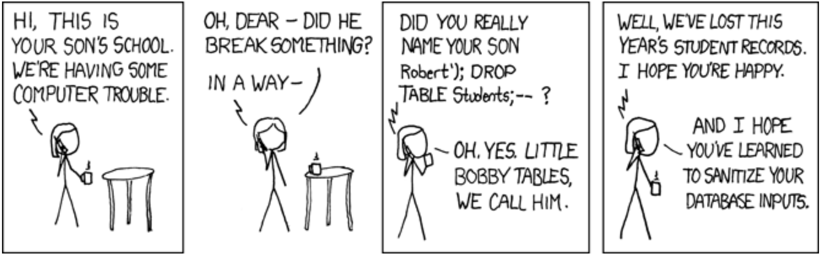
Object program may be ill-formed, and nesting is tricky.

Programs as data

```
String pancakes = "cook (mix [...], ...)";
double kcal = Nutrition.interpret(pancakes);
```

(cf. POSIX `regcomp`)

Object program may be ill-formed, and nesting is tricky.



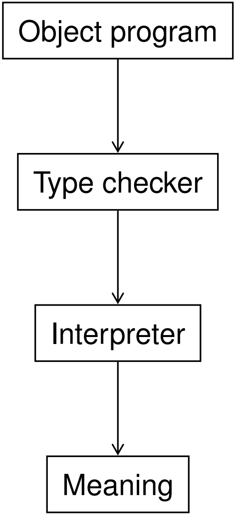
“Exploits of a mom”, http://xkcd.com/327/

Programs as data

```
Recipe pancakes = new Cook(new Mix(...), ...);
double kcal = Nutrition.interpret(pancakes);
```

Object program may be ill-typed or contain unbound variables, and we won’t find out until we actually generate it. Besides—

Tagging overhead



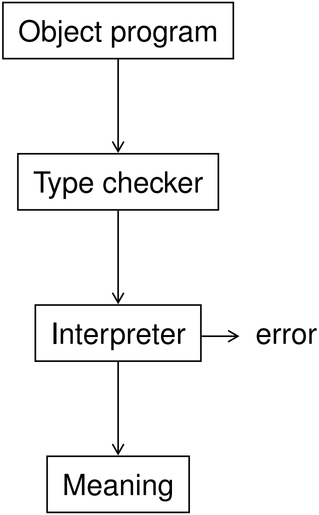
Object program

Type checker

Interpreter

Meaning

Tagging overhead


Object program

Type checker

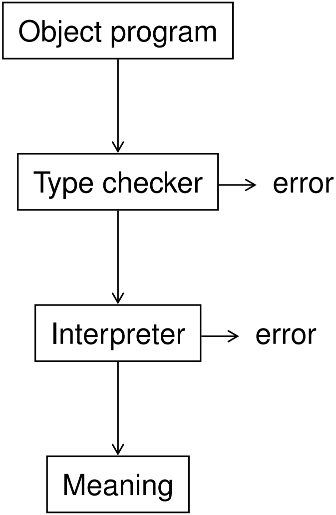
Interpreter

error

Meaning

Value use and environment lookup require dispatch that may fail.

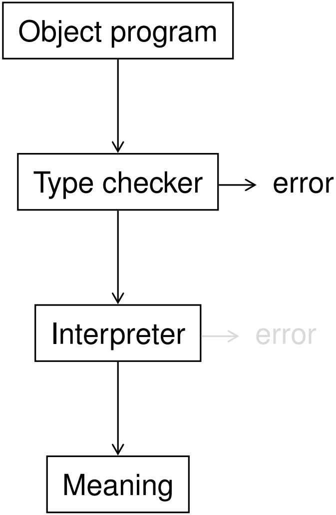
Tagging overhead



Well-typed programs
don’t go wrong.

Value use and environment lookup
require dispatch that may fail.

Tagging overhead



Well-typed programs
don't go wrong.

Type and binding safety in the
object language should be ensured
by the metalanguage.

Common subexpressions

Known-shared terms should be interpreted just once.

```
r = ... measure 1 teaspoon salt ...
    ... measure 1 teaspoon salt ...
```

Common subexpressions

Known-shared terms should be interpreted just once.

```
r = ... measure 1 teaspoon salt ...
    ... measure 1 teaspoon salt ...
```

```
Recipe s = new Measure(1, new Teaspoon(), new Salt());
Recipe r = ... s ... s ...;
double kcal = Nutrition.interpret(r);
```

Common subexpressions

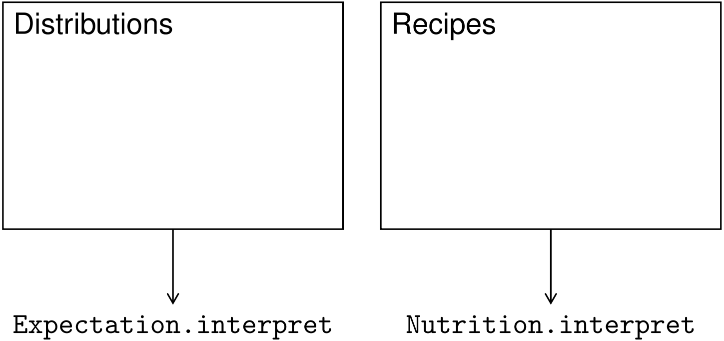
Known-shared terms should be interpreted just once.

```
r = ... measure 1 teaspoon salt ...
    ... measure 1 teaspoon salt ...
```

```
Recipe s = new Measure(1, new Teaspoon(), new Salt());
Recipe r = ... s ... s ...;
double kcal = Nutrition.interpret(r);
```

Sharing object values	Sharing object terms
Cook something once and use it many times	Cook the same thing many times
Make a decision once and use it many times	Make the same decision many times
Parse an input once and use it many times	Parse the same input format many times

Embedding interpreters



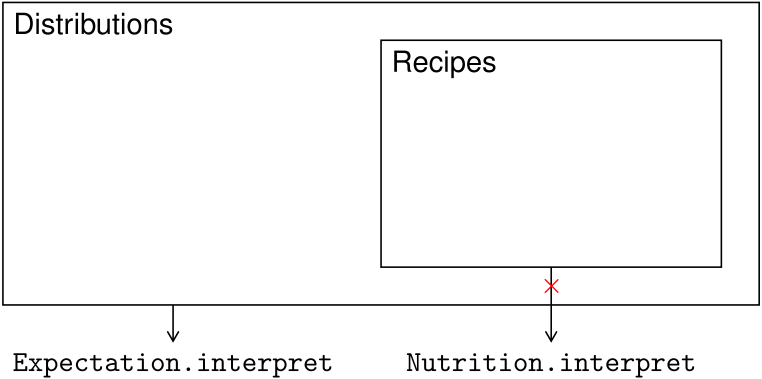
Distributions

Recipes

`Expectation.interpret`

`Nutrition.interpret`

Embedding interpreters



Container “map”/“functoriality”

Need to either make a “native call” to the nutrition interpreter, or port the nutrition interpreter into the distribution language.

Outline

- Representing knowledge as programs
  - Recursive syntactic structure
  - Multiple semantic interpretations
  - Binding and procedural abstraction
  - Types

- Embedding languages
  - Tagging overhead
  - Common subexpressions
  - Embedding interpreters

- ► **Preserving types and binding**
  - Finally tagless
  - Closing the stage

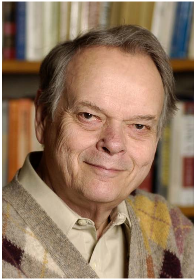
Programs as data

The central question:

How to represent object programs in the metalanguage?

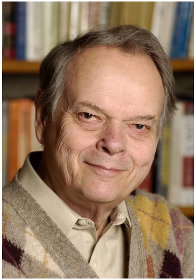
Desiderata:

- Multiple interpretations
- Preserve types and binding
- Preserve sharing
- Embed interpreters



It should also be possible to define languages, such as ALGOL 68, with a highly refined syntactic type structure. Ideally, such a treatment should be meta-circular . . .

(John Reynolds, 1972)



It should also be possible to define languages, such as ALGOL 68, with a highly refined syntactic type structure. Ideally, such a treatment should be meta-circular …

(John Reynolds, 1972)

Systems F and F_ω (Jean-Yves Girard, 1972)

Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur. Thèse de doctorat d'état, Université Paris VII.

Two ways to represent type and binding safety

1. "Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages."

   Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. APLAS 2007.
   Journal version submitted.

   Replace term constructors by interpreter branches.
   Payoffs (using generics over generics):

   - Eliminate tagging
   - Preserve sharing
   - Ease "native calling"
   - Interpret terms and types multiply

2. "Closing the stage: from staged code to typed closures."

   Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. PEPM 2008.

   Convert terms to closures with typed environments.

Two ways to represent type and binding safety

1. "Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages."

   Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. APLAS 2007.
   Journal version submitted.

   Replace term constructors by interpreter branches.

   Payoffs (using generics over generics):

   - Eliminate tagging
   - Preserve sharing
   - Ease "native calling"
   - Interpret terms and types multiply

2. "Closing the stage: from staged code to typed closures."

   Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. PEPM 2008.

   Convert terms to closures with typed environments.

Finally tagless

Just an abstract data type (Milner).

$$E ::= \texttt{oats} \mid \texttt{water} \mid \texttt{measure 1 cup}\ E \mid \texttt{mix}\ E\ E$$

```
interface Symantics<Repr> {
  Repr oats();  Repr water();
  Repr measure_1_cup(Repr e);
  Repr mix(Repr e1, Repr e2);
}
```

Finally tagless

Just an abstract data type (Milner).

$$E ::= \texttt{oats} \mid \texttt{water} \mid \texttt{measure 1 cup } E \mid \texttt{mix } E\ E$$

```
soaked_oats = mix (measure 1 cup oats)
                  (measure 1 cup water)
```

```
interface Symantics<Repr> {
  Repr oats();  Repr water();
  Repr measure_1_cup(Repr e);
  Repr mix(Repr e1, Repr e2);
}
```

```
<Repr> Repr soaked_oats(Symantics<Repr> s) {
  return s.mix(s.measure_1_cup(s.oats()),
               s.measure_1_cup(s.water()));
}
```

Finally tagless with binding safety

Meta-binding represents object binding (Washburn & Weirich).

$$\begin{aligned} E ::= {} & \texttt{oats} \mid \texttt{water} \mid \texttt{measure 1 cup } E \mid \texttt{mix } E\ E \\ & \mid x \mid \lambda x.\, E \mid E(E) \end{aligned}$$

```
soak = λx. mix x (measure 1 cup water)
```

```
interface Symantics<Repr> {
  Repr oats();  Repr water();
  Repr measure_1_cup(Repr e);
  Repr mix(Repr e1, Repr e2);
  Repr lambda({Repr => Repr} body);
  Repr apply(Repr fun, Repr arg);
}
<Repr> Repr soak(Symantics<Repr> s) {
  return s.lambda(Repr x =>
      s.mix(x, s.measure_1_cup(s.water())));
}
```

Finally tagless with type and binding safety

Meta-typing represents object typing (us).

$$\begin{aligned} E &::= \texttt{oats} \mid \texttt{water} \mid \texttt{measure 1 cup}\ E \mid \texttt{mix}\ E\ E \\ &\quad \mid x \mid \lambda x.\, E \mid E(E) \end{aligned}$$

$$T ::= \text{food} \mid T \to T$$

```
interface Symantics<Repr> {
  Repr oats();  Repr water();
  Repr measure_1_cup(Repr e);
  Repr mix(Repr e1, Repr e2);
  Repr lambda({Repr => Repr} body);
  Repr apply(Repr fun, Repr arg);
}
<Repr> Repr soak(Symantics<Repr> s) {
  return s.lambda(Repr x =>
      s.mix(x, s.measure_1_cup(s.water())));
}
```

Finally tagless with type and binding safety

Meta-typing represents object typing (us).

$$\begin{aligned} E &::= \texttt{oats} \mid \texttt{water} \mid \texttt{measure 1 cup}\ E \mid \texttt{mix}\ E\ E \\ &\quad \mid x \mid \lambda x.\,E \mid E(E) \\ T &::= \mathsf{food} \mid T \to T \end{aligned}$$

```
interface Symantics<Repr> {
  Repr       oats();  Repr       water();
  Repr       measure_1_cup(Repr       e);
  Repr       mix(Repr       e1, Repr       e2);
        Repr         lambda({Repr   => Repr   } body);
        Repr    apply(Repr         fun, Repr    arg);
}
<Repr> Repr               soak(Symantics<Repr> s) {
  return s.lambda(Repr       x =>
      s.mix(x, s.measure_1_cup(s.water())));
}
```

Finally tagless with type and binding safety

Meta-typing represents object typing (us).

$$\begin{aligned} E &::= \mathtt{oats} \mid \mathtt{water} \mid \mathtt{measure\ 1\ cup}\ E \mid \mathtt{mix}\ E\ E \\ &\quad\ \mid x \mid \lambda x.\, E \mid E(E) \\ T &::= \mathrm{food} \mid T \to T \end{aligned}$$

```
interface Symantics<Repr> {
  Repr<Food> oats();  Repr<Food> water();
  Repr<Food> measure_1_cup(Repr<Food> e);
  Repr<Food> mix(Repr<Food> e1, Repr<Food> e2);
  <A,B> Repr<{A=>B}> lambda({Repr<A>=>Repr<B>} body);
  <A,B> Repr<B> apply(Repr<{A=>B}> fun, Repr<A> arg);
}
<Repr> Repr<{Food=>Food}> soak(Symantics<Repr> s) {
  return s.lambda(Repr<Food> x =>
      s.mix(x, s.measure_1_cup(s.water())));
}
```

Two ways to represent type and binding safety

1. "Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages."

   Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. APLAS 2007.
   Journal version submitted.

   Replace term constructors by interpreter branches.
   Payoffs (using generics over generics):

   - Eliminate tagging
   - Preserve sharing
   - Ease "native calling"
   - Interpret terms and types multiply

2. "Closing the stage: from staged code to typed closures."

   Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. PEPM 2008.

   Convert terms to closures with typed environments.

Two ways to represent type and binding safety

1. “Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages.”

   Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. APLAS 2007.
   Journal version submitted.

   Replace term constructors by interpreter branches.
   Payoffs (using generics over generics):

   - Eliminate tagging
   - Preserve sharing
   - Ease “native calling”
   - Interpret terms and types multiply

2. “Closing the stage: from staged code to typed closures.”

   Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. PEPM 2008.

   Convert terms to closures with typed environments.

Two ways to represent type and binding safety

1. "Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages."

   Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. APLAS 2007. Journal version submitted.

   Replace term constructors by interpreter branches.
   Payoffs (using generics over generics):

   - Eliminate tagging
   - Preserve sharing
   - Ease "native calling"
   - Interpret terms and types multiply

2. "Closing the stage: from staged code to typed closures."

   Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. PEPM 2008.

   Convert terms to closures with typed environments.

Closing the stage

$$x \qquad \rightsquigarrow$$

Closing the stage

$$x, y \quad \vdash x \qquad \rightsquigarrow \qquad \lambda\langle x, y\rangle.\, x$$

Closing the stage

$$
\begin{array}{lll}
x, y & \vdash x & \rightsquigarrow \lambda\langle x, y\rangle.\ x \\
x, y & \vdash 3 & \rightsquigarrow \lambda\langle x, y\rangle.\ 3 \\
x, y & \vdash x + 3 & \rightsquigarrow \lambda\langle x, y\rangle.\ x + 3
\end{array}
$$

Closing the stage

$$
\begin{array}{lll}
x, y \vdash x & \rightsquigarrow & \lambda\langle x, y\rangle.\ x \\
x, y \vdash 3 & \rightsquigarrow & \lambda\langle x, y\rangle.\ 3 \\
x, y \vdash x + 3 & \rightsquigarrow & \lambda\langle x, y\rangle.\ x + 3 \\
 & & \\
x, y, z \vdash x & \rightsquigarrow & \lambda\langle x, y, z\rangle.\ x \\
x, y, z \vdash 3 & \rightsquigarrow & \lambda\langle x, y, z\rangle.\ 3 \\
x, y, z \vdash x + 3 & \rightsquigarrow & \lambda\langle x, y, z\rangle.\ x + 3
\end{array}
$$

Closing the stage

$$\begin{tikzcd}
x, y \vdash x \arrow[r, squiggly] & \lambda\langle x, y\rangle.\ x \arrow[ddd, bend left, "\text{coerce}"] \\
x, y \vdash 3 \arrow[r, squiggly] & \lambda\langle x, y\rangle.\ 3 \\
x, y \vdash x + 3 \arrow[r, squiggly] & \lambda\langle x, y\rangle.\ x + 3 \\
x, y, z \vdash x \arrow[r, squiggly] & \lambda\langle x, y, z\rangle.\ x \\
x, y, z \vdash 3 \arrow[r, squiggly] & \lambda\langle x, y, z\rangle.\ 3 \\
x, y, z \vdash x + 3 \arrow[r, squiggly] & \lambda\langle x, y, z\rangle.\ x + 3
\end{tikzcd}$$

Closing the stage

$$\begin{tikzcd}
x, y \vdash x \arrow[r, rightsquigarrow] & \lambda\langle x, y\rangle.\ x \\
x, y \vdash 3 \arrow[r, rightsquigarrow] & \lambda\langle x, y\rangle.\ 3 \arrow[ddd, bend left, "\text{coerce}"] \\
x, y \vdash x + 3 \arrow[r, rightsquigarrow] & \lambda\langle x, y\rangle.\ x + 3 \\
x, y, z \vdash x \arrow[r, rightsquigarrow] & \lambda\langle x, y, z\rangle.\ x \\
x, y, z \vdash 3 \arrow[r, rightsquigarrow] & \lambda\langle x, y, z\rangle.\ 3 \\
x, y, z \vdash x + 3 \arrow[r, rightsquigarrow] & \lambda\langle x, y, z\rangle.\ x + 3
\end{tikzcd}$$

Closing the stage

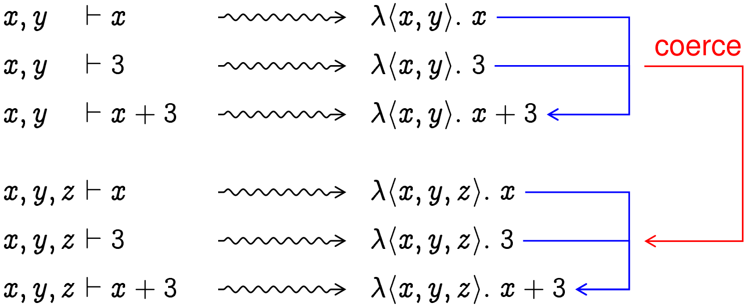
$$\begin{tikzcd}
x, y \vdash x \arrow[r, rightsquigarrow] & \lambda\langle x, y\rangle.\ x \\
x, y \vdash 3 \arrow[r, rightsquigarrow] & \lambda\langle x, y\rangle.\ 3 \\
x, y \vdash x + 3 \arrow[r, rightsquigarrow] & \lambda\langle x, y\rangle.\ x + 3 \arrow[ddd, bend left, "\text{coerce}"] \\
x, y, z \vdash x \arrow[r, rightsquigarrow] & \lambda\langle x, y, z\rangle.\ x \\
x, y, z \vdash 3 \arrow[r, rightsquigarrow] & \lambda\langle x, y, z\rangle.\ 3 \\
x, y, z \vdash x + 3 \arrow[r, rightsquigarrow] & \lambda\langle x, y, z\rangle.\ x + 3
\end{tikzcd}$$

Closing the stage

$$\begin{tikzcd}
x, y \vdash x \arrow[r, squiggly] & \lambda\langle x, y\rangle.\ x \arrow[dd, bend left] \\
x, y \vdash 3 \arrow[r, squiggly] & \lambda\langle x, y\rangle.\ 3 \arrow[d, bend left] \\
x, y \vdash x + 3 \arrow[r, squiggly] & \lambda\langle x, y\rangle.\ x + 3
\end{tikzcd}$$

$$\begin{tikzcd}
x, y, z \vdash x \arrow[r, squiggly] & \lambda\langle x, y, z\rangle.\ x \arrow[dd, bend left] \\
x, y, z \vdash 3 \arrow[r, squiggly] & \lambda\langle x, y, z\rangle.\ 3 \arrow[d, bend left] \\
x, y, z \vdash x + 3 \arrow[r, squiggly] & \lambda\langle x, y, z\rangle.\ x + 3
\end{tikzcd}$$

Closing the stage



$$x, y \vdash x \quad \rightsquigarrow \quad \lambda\langle x, y\rangle.\ x$$

$$x, y \vdash 3 \quad \rightsquigarrow \quad \lambda\langle x, y\rangle.\ 3$$

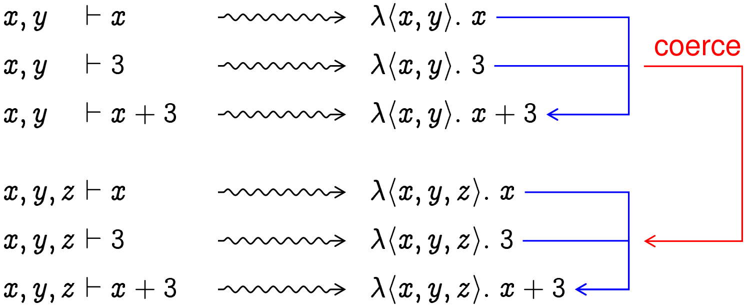
$$x, y \vdash x + 3 \quad \rightsquigarrow \quad \lambda\langle x, y\rangle.\ x + 3$$

$$x, y, z \vdash x \quad \rightsquigarrow \quad \lambda\langle x, y, z\rangle.\ x$$

$$x, y, z \vdash 3 \quad \rightsquigarrow \quad \lambda\langle x, y, z\rangle.\ 3$$

$$x, y, z \vdash x + 3 \quad \rightsquigarrow \quad \lambda\langle x, y, z\rangle.\ x + 3$$

Closing the stage



$x, y \vdash x \rightsquigarrow \lambda\langle x, y\rangle.\ x$

$x, y \vdash 3 \rightsquigarrow \lambda\langle x, y\rangle.\ 3$

$x, y \vdash x + 3 \rightsquigarrow \lambda\langle x, y\rangle.\ x + 3$

$x, y, z \vdash x \rightsquigarrow \lambda\langle x, y, z\rangle.\ x$

$x, y, z \vdash 3 \rightsquigarrow \lambda\langle x, y, z\rangle.\ 3$

$x, y, z \vdash x + 3 \rightsquigarrow \lambda\langle x, y, z\rangle.\ x + 3$

Encode binding in the object language
using tuples and generics in the metalanguage.

Conclusion

Represent knowledge as programs!

- It’s executable!
- It’s composable!
- It’s expressive!
- It’s intuitive!

Embedding languages in each other:
How to preserve types and binding?

- Replace term constructors by interpreter branches.
- Convert terms to closures with typed environments.