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

Embedding languages

Chung-chieh Shan
Oatmeal pancakes


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. How long does it take to make?
Reserve kitchen equipment. How many calories does it have?
Typeset it. How much does it cost?
Is it vegetarian? 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, ...], ...)
Interpreting recipes

Make it. How long does it take to make?
Reserve kitchen equipment. How many calories does it have?
Typeset it. How much does it cost?
Is it vegetarian? 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

cook

mix

soak

3/4 cup measure

1 cup whole_wheat_flour

measure

3/4 cup oats

...
Recursive syntactic structure

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

\[ U ::= \text{cup} \mid \text{tablespoon} \mid \text{teaspoon} \mid \ldots \]
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.

\[
\begin{align*}
\text{[oats]} &= 300 \text{ kcal} : 1 \text{ cup} : 80 \text{ gram} \\
\text{[water]} &= 0 \text{ kcal} : 1 \text{ cup} : 230 \text{ gram} \\
\text{[measure } n \text{ cup } E]\] &= \frac{xn}{y} \text{ kcal} : n \text{ cup} : \frac{zn}{y} \text{ gram} \\
\text{where } [E] &= x \text{ kcal} : y \text{ cup} : z \text{ gram}
\end{align*}
\]

\[
\text{[mix } [E_1, \ldots, E_n]\] = \sum_{i=1}^{n} [E_i]
\]

(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.

\[ T ::= \text{food} \mid \text{number} \mid \text{unit} \mid T \text{ list} \mid T \rightarrow T \]
Types

Classify terms more finely.

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

\[
\begin{align*}
[x : T_1] \\
\vdots \\
E : T_2 \\
\text{water : food} \\
\lambda x. E : T_1 \to T_2 \\
E_1 : T_1 \to T_2 \\
E_2 : T_1 \\
E_1(E_2) : T_2
\end{align*}
\]
Types

Classify terms more finely.

\[ T ::= \text{food} \mid \text{number} \mid \text{unit} \mid T \text{ list} \mid T \rightarrow T \]

\[
[x : T_1] \\
:\vdash \\
E : T_2 \\
E_1 : T_1 \rightarrow T_2 \\
E_2 : T_1 \\
\]

\[
\text{water} : \text{food} \\
\lambda x. E : T_1 \rightarrow T_2 \\
E_1(E_2) : T_2 \\
\text{seasoning} : \text{food} \\
\text{soak} : \text{number} \rightarrow \text{unit} \rightarrow \text{food} \rightarrow \text{food}
\]
Types

Classify terms more finely.

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

\[
\begin{align*}
[x : T_1] & \\
\vdots & \\
E : T_2 & \\
E_1 : T_1 \to T_2 & \quad E_2 : T_1 \\
\hline
\text{water} : \text{food} & \quad \lambda x. E : T_1 \to T_2 & \quad E_1(E_2) : T_2 \\
\text{seasoning} : \text{food} & \quad \text{soak} : \text{number} \to \text{unit} \to \text{food} \to \text{food}
\end{align*}
\]

Further distinctions: mass (oats) vs count (pancakes), carnivore (bone sugar) vs vegetarian (cane sugar), \ldots

Static safety guarantees.
Types

Classify terms more finely.

\[ T ::= \text{food} \mid \text{number} \mid \text{unit} \mid T \ 	ext{list} \mid T \rightarrow T \]

\[ [x : T_1] \]

\[ \vdash \]

\[ E : T_2 \]

\[ E_1 : T_1 \rightarrow T_2 \]

\[ E_2 : T_1 \]

\[ \text{water} : \text{food} \quad \lambda x. E : T_1 \rightarrow T_2 \quad E_1(E_2) : T_2 \]

\[ \text{seasoning} : \text{food} \quad \text{soak} : \text{number} \rightarrow \text{unit} \rightarrow \text{food} \rightarrow \text{food} \]

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 $\approx$ what the modeled agent knows
  Object terms $\approx$ 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 $\approx$ what the modeled agent knows
  Object terms $\approx$ what the modeling agent believes
- *mixed quotation*:
  Bush also said his administration would “achieve our objectives” in Iraq. (New York Times, November 4, 2004)


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/
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

 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

Object program

Type checker → error

Interpreter → error

Meaning

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.

\[
\begin{align*}
  r &= \ldots \text{measure 1 teaspoon salt} \ldots \\
     &\quad \ldots \text{measure 1 teaspoon salt} \ldots
\end{align*}
\]
Common subexpressions

Known-shared terms should be interpreted just once.

\[ r = \ldots \text{measure 1 teaspoon salt} \ldots \]
\[ \ldots \text{measure 1 teaspoon salt} \ldots \]

Recipe \( s = \text{new Measure}(1, \text{new Teaspoon}(), \text{new Salt}()); \)
Recipe \( r = \ldots s \ldots s \ldots; \)
double \( \text{kcal} = \text{Nutrition.interpret}(r); \)
Common subexpressions

Known-shared terms should be interpreted just once.

\[
\begin{align*}
\text{Recipe } s & = \text{new Measure}(1, \text{new Teaspoon}(), \text{new Salt}()); \\
\text{Recipe } r & = \ldots \text{ } s \ldots \text{ } s \ldots; \\
\text{double } \text{kcal} & = \text{Nutrition.interpret}(r);
\end{align*}
\]

<table>
<thead>
<tr>
<th>Sharing object <strong>values</strong></th>
<th>Sharing object <strong>terms</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cook something once and use it many times</td>
<td>Cook the same thing many times</td>
</tr>
<tr>
<td>Make a decision once and use it many times</td>
<td>Make the same decision many times</td>
</tr>
<tr>
<td>Parse an input once and use it many times</td>
<td>Parse the same input format many times</td>
</tr>
</tbody>
</table>


Embedding interpreters

Distributions

Recipes

Expectation.interpret

Nutrition.interpret

Need to either make a "native call" to the nutrition interpreter, or port the nutrition interpreter into the distribution language.
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
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 $F_\omega$ (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.”

   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.”

   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).

\[
E ::= \text{oats} | \text{water} | \text{measure 1 cup } E | \text{mix } E \ E
\]

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

```java
Repr lambda({Repr => Repr} body);
Repr apply(Repr fun, Repr arg);
```
Finally tagless

Just an abstract data type (Milner).

\[ E ::= \text{oats} | \text{water} | \text{measure 1 cup} \, E | \text{mix} \, E \, E \]

\[
\text{soaked_oats} = \text{mix} (\text{measure 1 cup oats}) \\
\hspace{1em} (\text{measure 1 cup water})
\]

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).

\[ \begin{align*}
E &::= \text{oats} \mid \text{water} \mid \text{measure 1 cup} \ E \mid \text{mix} \ E \ E \\
&\mid x \mid \lambda x.\ E \mid E(E)
\end{align*} \]

soak = \lambda x.\ \text{mix} \ x \ (\text{measure 1 cup} \ \text{water})

```java
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).

\[
E ::= \text{oats} | \text{water} | \text{measure 1 cup} \ E | \text{mix} \ E \ E
\]
\[
| x | \lambda x. \ E | \ E(\ E)
\]

\[
T ::= \text{food} | T \to T
\]

```java
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).

\[ E ::= \text{oats} \mid \text{water} \mid \text{measure 1 cup } E \mid \text{mix } E \ E \mid x \mid \lambda x. E \mid E(E) \]

\[ T ::= \text{food} \mid T \rightarrow T \]

```java
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).

\[
E ::= \text{oats} \mid \text{water} \mid \text{measure 1 cup } E \mid \text{mix } E \ E \\
    \mid x \mid \lambda x. \ E \mid E(E)
\]

\[
T ::= \text{food} \mid T \rightarrow T
\]

```java
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.”
   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.”

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.”

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

\[ x \quad \rightarrow \]
Closing the stage

\[ x, y \vdash x \quad \rightarrow \quad \lambda \langle x, y \rangle . x \]
Closing the stage

\[ x, y \vdash x \quad \Rightarrow \quad \lambda \langle x, y \rangle . x \]
\[ x, y \vdash 3 \quad \Rightarrow \quad \lambda \langle x, y \rangle . 3 \]
\[ x, y \vdash x + 3 \quad \Rightarrow \quad \lambda \langle x, y \rangle . x + 3 \]
Closing the stage

\[ x, y \vdash x \quad \rightarrow \quad \lambda\langle x, y \rangle. \ x \]
\[ x, y \vdash 3 \quad \rightarrow \quad \lambda\langle x, y \rangle. \ 3 \]
\[ x, y \vdash x + 3 \quad \rightarrow \quad \lambda\langle x, y \rangle. \ x + 3 \]

\[ x, y, z \vdash x \quad \rightarrow \quad \lambda\langle x, y, z \rangle. \ x \]
\[ x, y, z \vdash 3 \quad \rightarrow \quad \lambda\langle x, y, z \rangle. \ 3 \]
\[ x, y, z \vdash x + 3 \quad \rightarrow \quad \lambda\langle x, y, z \rangle. \ x + 3 \]
Closing the stage

\[
\begin{align*}
\Gamma & : x \quad \vdash x \quad \Rightarrow \quad \lambda \langle x, y \rangle \cdot x \\
\Gamma & : x, y \quad \vdash 3 \quad \Rightarrow \quad \lambda \langle x, y \rangle \cdot 3 \\
\Gamma & : x, y \quad \vdash x + 3 \quad \Rightarrow \quad \lambda \langle x, y \rangle \cdot (x + 3) \\
\Gamma & : x, y, z \quad \vdash x \quad \Rightarrow \quad \lambda \langle x, y, z \rangle \cdot x \\
\Gamma & : x, y, z \quad \vdash 3 \quad \Rightarrow \quad \lambda \langle x, y, z \rangle \cdot 3 \\
\Gamma & : x, y, z \quad \vdash x + 3 \quad \Rightarrow \quad \lambda \langle x, y, z \rangle \cdot (x + 3)
\end{align*}
\]
Closing the stage

\[ x, y \vdash x \quad \mapsto \quad \lambda\langle x, y\rangle . x \]

\[ x, y \vdash 3 \quad \mapsto \quad \lambda\langle x, y\rangle . 3 \]

\[ x, y \vdash x + 3 \quad \mapsto \quad \lambda\langle x, y\rangle . x + 3 \]

\[ x, y, z \vdash x \quad \mapsto \quad \lambda\langle x, y, z\rangle . x \]

\[ x, y, z \vdash 3 \quad \mapsto \quad \lambda\langle x, y, z\rangle . 3 \]

\[ x, y, z \vdash x + 3 \quad \mapsto \quad \lambda\langle x, y, z\rangle . x + 3 \]
Closing the stage

\[
x, y \vdash x \quad \rightsquigarrow \quad \lambda(x, y). \, x
\]
\[
x, y \vdash 3 \quad \rightsquigarrow \quad \lambda(x, y). \, 3
\]
\[
x, y \vdash x + 3 \quad \rightsquigarrow \quad \lambda(x, y). \, x + 3
\]
\[
x, y, z \vdash x \quad \rightsquigarrow \quad \lambda(x, y, z). \, x
\]
\[
x, y, z \vdash 3 \quad \rightsquigarrow \quad \lambda(x, y, z). \, 3
\]
\[
x, y, z \vdash x + 3 \quad \rightsquigarrow \quad \lambda(x, y, z). \, x + 3
\]
Closing the stage

\[
\begin{align*}
  x, y & \vdash x & \quad \Rightarrow & \quad \lambda \langle x, y \rangle. \, x \\
  x, y & \vdash 3 & \quad \Rightarrow & \quad \lambda \langle x, y \rangle. \, 3 \\
  x, y & \vdash x + 3 & \quad \Rightarrow & \quad \lambda \langle x, y \rangle. \, x + 3 \\
  x, y, z & \vdash x & \quad \Rightarrow & \quad \lambda \langle x, y, z \rangle. \, x \\
  x, y, z & \vdash 3 & \quad \Rightarrow & \quad \lambda \langle x, y, z \rangle. \, 3 \\
  x, y, z & \vdash x + 3 & \quad \Rightarrow & \quad \lambda \langle x, y, z \rangle. \, x + 3
\end{align*}
\]
Closing the stage

\[
\begin{align*}
  x, y &\vdash x & \mapsto & \lambda\langle x, y \rangle. x \\
  x, y &\vdash 3 & \mapsto & \lambda\langle x, y \rangle. 3 \\
  x, y &\vdash x + 3 & \mapsto & \lambda\langle x, y \rangle. x + 3 \\
  x, y, z &\vdash x & \mapsto & \lambda\langle x, y, z \rangle. x \\
  x, y, z &\vdash 3 & \mapsto & \lambda\langle x, y, z \rangle. 3 \\
  x, y, z &\vdash x + 3 & \mapsto & \lambda\langle x, y, z \rangle. x + 3
\end{align*}
\]
Closing the stage

\[
x, y \vdash x \quad \longrightarrow \quad \lambda(x, y). x
\]
\[
x, y \vdash 3 \quad \longrightarrow \quad \lambda(x, y). 3
\]
\[
x, y \vdash x + 3 \quad \longrightarrow \quad \lambda(x, y). x + 3
\]
\[
x, y, z \vdash x \quad \longrightarrow \quad \lambda(x, y, z). x
\]
\[
x, y, z \vdash 3 \quad \longrightarrow \quad \lambda(x, y, z). 3
\]
\[
x, y, z \vdash x + 3 \quad \longrightarrow \quad \lambda(x, y, z). x + 3
\]

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.