A Practical, Typed Variant Object Model
Or, How to Stand On Your Head and Enjoy the View

Pottayil Harisanker Menon, Zachary Palmer, Scott F. Smith, Alexander Rozenshteyn

The Johns Hopkins University

October 22, 2012
Object Encodings

- Record-Based Encodings
Object Encodings

- Record-Based Encodings
  - Foundation for traditional OO languages
Object Encodings

- Record-Based Encodings
  - Foundation for traditional OO languages
  - Easier to type

- Variant-Based Encodings
  - Actor-based languages (Erlang)
  - Harder to type
Object Encodings

- Record-Based Encodings
  - Foundation for traditional OO languages
  - Easier to type
  - Common [Cardelli '84] [Cook '89] . . .
Object Encodings

- Record-Based Encodings
  - Foundation for traditional OO languages
  - Easier to type
  - Common [Cardelli ’84] [Cook ’89] . . .

- Variant-Based Encodings
Object Encodings

- **Record-Based Encodings**
  - Foundation for traditional OO languages
  - Easier to type
  - Common [Cardelli ’84] [Cook ’89] . . .

- **Variant-Based Encodings**
  - Actor-based languages (Erlang)
Object Encodings

- Record-Based Encodings
  - Foundation for traditional OO languages
  - Easier to type
  - Common [Cardelli ’84] [Cook ’89] . . .

- Variant-Based Encodings
  - Actor-based languages (Erlang)
  - Harder to type
Record-Based Object Encoding

(Scala)

```scala
1 object a {
2 }
```

(OCaml)

```ocaml
1 let a = {
2 }
```
Record-Based Object Encoding

**Object fields are record fields**
Object fields are record fields

Methods are fields with functions
Object fields are record fields
Methods are fields with functions
Invocation projects methods
Record-Based Object Encoding

**Scala**

```scala
object a {
  val v = 5
  def mth(x: Int) : Int = { x+v }
  def foo(x: Unit){}
}

a.mth(3)
```

**OCaml**

```ocaml
let a = {
  v = ref 5;
  mth = fun self ->
    fun x -> x+!self.v
  foo = fun () -> ()
} in

a.mth a 3
```

- Object fields are record fields
- Methods are fields with functions
- Invocation projects methods
  - We ignore self-hiding for now.
Duality

Variants ⇔ Records
Variant-Based Encoding

(Scala)

```scala
object a {
  ...
}
```

(OCaml)

```ocaml
let a = fun msg ->
  match msg with
  ...
```
Variant-Based Encoding

**(Scala)**

```scala
object a {
  val v = 5
}
```

**(OCaml)**

```ocaml
let v = ref 5 in
let a = fun msg ->
  match msg with
  ...
```

- Fields by closure
Variant-Based Encoding

(Scala)

```scala
object a {
  val v = 5
  def mth(x: Int) : Int = { x + v }
  def foo(x: Unit){}
}
```

(OCaml)

```ocaml
let v = ref 5 in
let a = fun msg ->
  match msg with
  | 'mth (self, x) ->
    x + !self.v
  | 'foo () -> ()
```

- Fields by closure
- Methods are other message handlers
Variant-Based Encoding

(Scala)
```scala
object a {
  val v = 5
  def mth(x: Int): Int = { x + v }
  def foo(x: Unit) {}
}

a.mth(3)
```

(OCaml)
```ocaml
let v = ref 5 in
let a = fun msg ->
  match msg with
  | 'mth (self, x) -> x + ! self.v
  | 'foo () -> ()
in a ('mth (a, 3))
```

- Fields by closure
- Methods are other message handlers
- Invocation is just message passing
Variant-Based Encoding

(Scala)

```scala
object a {
  val v = 5
  def mth(x: Int): Int = { x + v }
  def foo(x: Unit) {};
}

a.mth(3)
```

(OCaml)

```ocaml
let v = ref 5 in
let a = fun msg ->
  match msg with
  | 'mth (self, x) -> x + !self.v
  | 'foo () -> ()
in a ('mth (a, 3))
```

- Fields by closure
- Methods are other message handlers
- Invocation is just message passing
- But this doesn’t typecheck!
Typing Variant Destruction

1 \texttt{match} \ v \ \texttt{with}
2 \quad | \quad \textquoteleft\text{Odd} \ y \ -> \ y \ \text{mod} \ 2 \ = \ 1
3 \quad | \quad \textquoteleft\text{Dbl} \ x \ -> \ x \ + \ x

- Typechecking variant destruction is tricky
Typing Variant Destruction

1 \texttt{match \ v \ with}
2 \quad | \ ‘\texttt{Odd} \ y \ \rightarrow \ y \mod 2 = 1
3 \quad | \ ‘\texttt{Dbl} \ x \ \rightarrow \ x + x

- Typechecking variant destruction is tricky
- Most languages (e.g. Caml) fail on unification
Typing Variant Destruction

1 `match v with
2  | ‘Odd y -> y mod 2 = 1 : int ∪ bool
3  | ‘Dbl x -> x + x`

- Typechecking variant destruction is tricky
- Most languages (e.g. Caml) fail on unification
- Union types
Typing Variant Destruction

1 \texttt{match } 'Dbl 2 \texttt{ with}
2    \mid 'Odd y \to y \mod 2 = 1 : \texttt{int } \cup \texttt{ bool}
3    \mid 'Dbl x \to x + x

- Typechecking variant destruction is tricky
- Most languages (e.g. Caml) fail on unification
- Union types are insufficient!
Typing Variant Destruction

1 \texttt{match ‘Db1 2 with}
2    | ‘Odd y \rightarrow y \mod 2 = 1 : \texttt{int!}
3    | ‘Db1 x \rightarrow x + x

- Typechecking variant destruction is tricky
- Most languages (e.g. Caml) fail on unification
- Union types are insufficient!
- Record construction is heterogeneously typed
Typing Variant Destruction

1 \texttt{match} \ ‘\texttt{Dbl} 2 \ with
2 \quad \mid \ ‘\texttt{Odd} y \ \rightarrow \ y \ mod \ 2 = 1 : \texttt{int}!
3 \quad \mid \ ‘\texttt{Dbl} x \ \rightarrow \ x + x

- Typechecking variant destruction is tricky
- Most languages (e.g. Caml) fail on unification
- Union types are insufficient!
- Record construction is heterogeneously typed
- Variant destruction is not
Typing the Variant Encoding

Our objective: a purely type-inferred variant-based object encoding
Typing the Variant Encoding

Our objective: a purely type-inferred variant-based object encoding

This can work! We just need...
Typing the Variant Encoding

Our objective: a purely type-inferred variant-based object encoding

This can work! We just need...

- A couple new expression forms
- Weakly dependent types
- Precise polymorphism
- A whole-program typechecking pass
Typing the Variant Encoding

Our objective: a purely type-inferred variant-based object encoding

This can work! We just need...

- A couple new expression forms
- Weakly dependent types
- Precise polymorphism
- A whole-program typechecking pass

...and then we reap the benefits!
How We Get It: TinyBang

&

Onions

(Extensible, type-indexed records)
How We Get It: TinyBang

&

Onions

(Extensible, type-indexed records)

&

χ −> Scapes

(Functions with built-in patterns)
How We Get It: TinyBang

&

Onions
(Extensible, type-indexed records)

+  

\( \chi \rightarrow \)

Scapes
(Functions with built-in patterns)
Variant-Based Object Encoding

TinyBang

1  'dbl  x -> x + x

- Methods are scapes
Variant-Based Object Encoding

TinyBang

1. `\texttt{db1 \, x} \rightarrow x + x`

- Methods are scapes: \texttt{functions with patterns}
Variant-Based Object Encoding

**TinyBang**

\[ (\texttt{\textquotesingle dbl\ x -> x + x}) \texttt{\textquotesingle dbl\ 3} \]

- Methods are scapes: functions with patterns
- Invoke methods by passing messages
Variant-Based Object Encoding

TinyBang

1 (‘dbl x -> x + x) 'dbl 3

- Methods are scapes: functions with patterns
- Invoke methods by passing first-class messages
TinyBang

1 (`dbl x -> x + x) 'dbl 3

- Methods are scapes: functions with patterns
- Invoke methods by passing first-class messages (just labeled data)
Many Methods: Onioning Scapes

\`
\begin{align*}
1 & \texttt{dbl} \quad x & \rightarrow & x + x
\end{align*}
\`
Many Methods: Onioning Scapes

1. (`dbl x -> x + x`) &
2. (`odd y -> y mod 2 == 1`)

• Scapes are combined by *onioning*
Many Methods: Onioning Scapes

1 \((\texttt{'}dbl\ x \rightarrow x + x)\ \&\)
2 \((\texttt{'}odd\ y \rightarrow y \ mod\ 2 == 1)\) \(\texttt{'}dbl\ 2\)

- Scapes are combined by \textit{onioning}
- Application finds match
Many Methods: Onioning Scapes

1 \((\texttt{`dbl } x \rightarrow x + x) \land\)
2 \((\texttt{`odd } y \rightarrow y \mod 2 == 1))\) \((\texttt{`dbl } 2)\)

1 \texttt{object a \{}
2 \texttt{ def dbl(x:Int):Int = \{ x + x \}}
3 \texttt{ def pos(y:Int):Boolean = \{ y \% 2 == 1 \}}
4 \texttt{\}}
5 \texttt{a.db(2)}
Many Methods: Onioning Scapes

1. \((('\text{dbl } x \rightarrow x + x) \& ('\text{odd } y \rightarrow y \mod 2 == 1)) ('\text{dbl } 2)\) ⇒ 4

- Scapes are combined by \textit{onioning}
- Application finds \textit{rightmost} match (asymmetric)
Many Methods: Onioning Scapes

\[ ((\texttt{dbl} \ x \rightarrow x + x) \ & \ (\texttt{odd} \ y \rightarrow y \ mod \ 2 == 1)) \ (\texttt{dbl} \ 2) \ \Rightarrow \ 4 \]

- Scapes are combined by onioning.
- Application finds rightmost match (asymmetric).
- Subsumes case expressions.
Many Methods: Onioning Scapes

1 \(((\texttt{dbl } x \rightarrow x + x) \&
\begin{align*}
2 \hspace{1em} (\texttt{odd } y \rightarrow y \mod 2 == 1) & (\texttt{dbl } 2) \\
\end{align*}
\\Rightarrow \text{4}
\)

- Scapes are combined by \textit{onioning}
- Application finds rightmost match (asymmetric)
- Subsumes case expressions
- Generalizes First-Class Cases [Blume et. al. ’06]
Typing the Onion

1. (‘dbl $x \rightarrow x + x$) &
2. (‘odd $y \rightarrow y \mod 2 == 1$)

(‘dbl $\text{int} \cup ‘\text{odd int}) \rightarrow (\text{int} \cup \text{bool})$

- Simple union type loses alignment
Typing the Onion

1. `(dbl x -> x + x) &
2. (odd y -> y mod 2 == 1)

(`dbl int -> int) & (odd int -> bool)

- Simple union type loses alignment
- Onion type does not
Typing the Onion

1. (‘dbl x -> x + x) &
2. (‘odd y -> y mod 2 == 1)

(‘dbl int -> int) & (‘odd int -> bool)

- Simple union type loses alignment
- *Onion* type does not
- Weakly dependent type
Typing the Onion

\[
1. \ ('\text{dbl } x \rightarrow x + x) \land \\
2. \ ('\text{odd } y \rightarrow y \ mod \ 2 = 1)
\]

\[
('\text{dbl int } \rightarrow \text{int}) \land ('\text{odd int } \rightarrow \text{bool})
\]

- Simple union type loses alignment
- \textit{Onion type} does not
- Weakly dependent type
- Relies heavily on polymorphism
Fields

- Pure variant model: get/set messages
Fields

1. (‘dbl x -> x + x) &
2. (‘odd y -> y mod 2 == 1) &
3. ‘Z 5

- Pure variant model: get/set messages
- Hybrid model: variant methods, record fields
Fields

1 (‘dbl x -> x + x) &
2 (‘odd y -> y mod 2 == 1) &
3 ‘Z 5

- Pure variant model: get/set messages
- Hybrid model: variant methods, record fields
- Similar to type-indexed rows [Shields, Meijer ’01]
Fields

Pure variant model: get/set messages
Hybrid model: variant methods, record fields
Similar to type-indexed rows [Shields, Meijer ’01]
Labels implicitly create cells
Fields

```haskell
1 def o = ('dbl x -> x + x) &
2 ('odd y -> y mod 2 == 1) &
3 'Z 5
4 in o.Z
```

- Pure variant model: get/set messages
- Hybrid model: variant methods, record fields
- Similar to type-indexed rows [Shields, Meijer ’01]
- Labels implicitly create cells
- Field access by projection
Fields

```plaintext
def o = ('dbl x -> x + x) &
    ('odd y -> y mod 2 == 1) &
    'Z 5
in ('Z z -> z) o
```

- Pure variant model: get/set messages
- Hybrid model: variant methods, record fields
- Similar to type-indexed rows [Shields, Meijer ’01]
- Labels implicitly create cells
- Field access by projection/pattern match
Fields

```
1 def o = ('dbl x -> x + x) &
2      ('odd y -> y mod 2 == 1) &
3          'Z 5
4 in ('Z z -> z) o
```

- Pure variant model: get/set messages
- Hybrid model: variant methods, record fields
- Similar to type-indexed rows [Shields, Meijer '01]
- Labels implicitly create cells
- Field access by projection/pattern match
- But what about self?
Naïve Self

def ticker =
  'x 0 &
  ('inc _ ->
    self.x = self.x + 1 in self.x)
  in ticker 'inc ()
Naïve Self

```python
def ticker =
  'x 0 &
  ('inc _ & 'self self ->
    self.x = self.x + 1 in self.x)
in ticker 'inc ()
```

- Add 'self to all parameters
Naïve Self

```python
def ticker =
  `x 0 &
  (\textcolor{yellow}{`inc} \_ \& \textcolor{red}{`self self} ->
    self.x = self.x + 1 \text{ in self.x})
  in ticker `inc ()
```

- Add `self to all parameters
  - & is pattern conjunction
Naïve Self

```python
def ticker =
    'x 0 &
    ('inc _ & 'self self ->
        self.x = self.x + 1 in self.x)
in ticker ('inc () & 'self ticker)
```

- Add 'self to all parameters
- & is pattern conjunction
- Add 'self to all call sites
Naïve Self

```python
def ticker =
  `x 0 &
  (`inc _ & `self self ->
    self.x = self.x + 1 in self.x)
in ticker (`inc () & `self ticker)
```

- Add `self to all parameters
  - & is pattern conjunction
- Add `self to all call sites
- Be happy?
Naïve Self: Type Problems

```python
def obj =
    if something then
        ('foo _ & 'self s -> s 'bar () &
         ('bar _ -> 1)
    else
        ('foo _ & 'self s -> s 'baz () &
         ('baz _ -> 2)
    in obj 'foo ()
```
Naïve Self: Problems

\[
\alpha_{\text{SELF}} =
\]
\[
\left( \text{`foo }_ \& \text{ `self } \alpha_1 \to \text{ int} \right) \land
\left( \text{`bar } \to \text{ int} \right)
\]
where \( \alpha_1 \) has `bar
\[
\cup
\]
\[
\left( \text{`foo }_ \& \text{ `self } \alpha_2 \to \text{ int} \right) \land
\left( \text{`baz } \to \text{ int} \right)
\]
where \( \alpha_2 \) has `baz
Naïve Self: Problems

\[
\alpha_{\text{SELF}} \ll (\text{`foo } \& \text{ `self } \alpha_1 \rightarrow \text{ int}) \land \\
\text{ where } \alpha_1 \text{ has `bar}
\]

\[
\alpha_{\text{SELF}} \ll (\text{`foo } \& \text{ `self } \alpha_2 \rightarrow \text{ int}) \land \\
\text{ where } \alpha_2 \text{ has `baz}
\]
Self Solutions

- Classic object encodings [Bruce et. al. ’98]
Self Solutions

- Classic object encodings [Bruce et. al. '98]
  - Type of self is fixed at instantiation
Self Solutions

- Classic object encodings [Bruce et. al. ’98]
  - Type of self is fixed at instantiation
  - No object extensibility
Self Solutions

- Classic object encodings [Bruce et. al. ’98]
  - Type of self is fixed at instantiation
  - No object extensibility

- Extensible Object Calculus [Fisher et. al. ’98]
Self Solutions

- Classic object encodings [Bruce et. al. ’98]
  - Type of self is fixed at instantiation
  - No object extensibility

- Extensible Object Calculus [Fisher et. al. ’98]
  - Prototype objects: extensible but not callable
Self Solutions

- **Classic object encodings** [Bruce et. al. ’98]
  - Type of self is fixed at instantiation
  - No object extensibility

- **Extensible Object Calculus** [Fisher et. al. ’98]
  - Prototype objects: extensible but not callable
  - Proper objects: callable but not extensible
Self Solutions

- Classic object encodings [Bruce et. al. ’98]
  - Type of self is fixed at instantiation
  - No object extensibility

- Extensible Object Calculus [Fisher et. al. ’98]
  - Prototype objects: extensible but not callable
  - Proper objects: callable but not extensible
  - Prototypes can be sealed into proper objects
Self Solutions

- Classic object encodings [Bruce et. al. ’98]
  - Type of self is fixed at instantiation
  - No object extensibility

- Extensible Object Calculus [Fisher et. al. ’98]
  - Prototype objects: extensible but not callable
  - Proper objects: callable but not extensible
  - Prototypes can be sealed into proper objects
  - Sealing is permanent
Self Solutions

- Classic object encodings [Bruce et. al. ’98]
  - Type of self is fixed at instantiation
  - No object extensibility

- Extensible Object Calculus [Fisher et. al. ’98]
  - Prototype objects: extensible but not callable
  - Proper objects: callable but not extensible
  - Prototypes can be sealed into proper objects
  - Sealing is permanent
  - Sealing is meta-theoretic
Self Solutions

- **Classic object encodings** [Bruce et. al. ’98]
  - Type of self is fixed at instantiation
  - No object extensibility

- **Extensible Object Calculus** [Fisher et. al. ’98]
  - Prototype objects: extensible but not callable
  - Proper objects: callable but not extensible
  - Prototypes can be sealed into proper objects
  - Sealing is permanent
  - Sealing is meta-theoretic

- TinyBang
Self Solutions

- Classic object encodings [Bruce et. al. ’98]
  - Type of self is fixed at instantiation
  - No object extensibility

- Extensible Object Calculus [Fisher et. al. ’98]
  - Prototype objects: extensible but not callable
  - Proper objects: callable but not extensible
  - Prototypes can be sealed into proper objects
  - Sealing is permanent
  - Sealing is meta-theoretic

- TinyBang
  - Sealing is encodable (no meta-theory)
Self Solutions

- **Classic object encodings** [Bruce et. al. ’98]
  - Type of self is fixed at instantiation
  - No object extensibility
- **Extensible Object Calculus** [Fisher et. al. ’98]
  - Prototype objects: extensible but not callable
  - Proper objects: callable but not extensible
  - Prototypes can be *sealed* into proper objects
  - Sealing is permanent
  - Sealing is meta-theoretic
- **TinyBang**
  - Sealing is encodable (no meta-theory)
  - Sealed objects can be extended and resealed
def rec seal = obj ->
  obj &
  (msg -> obj (‘self (seal obj) & msg)) in
def point =
  ‘x 2 & ‘y 4 &
  (‘l1 _ & ‘self self -> self.x + self.y) in
def sealedPoint = seal point in
sealedPoint ‘l1 ()
...
Resealing Objects

```
... 

def obj = seal ( 
    'x 0 &
    ('inc _ & 'self self -> 
        self.x = self.x + 1 in self.x)) in
obj 'inc (); obj 'inc ();
def extobj = seal ( 
    obj &
    ('dbl _ & 'self self -> 
        self.x = self.x + self.x in self.x)) in
extobj 'dbl (); extobj 'inc ()

x = 0
```
Resealing Objects

```
...  

def obj = seal (  
    'x 0 &  
    ('inc _ & 'self self ->  
      self.x = self.x + 1 in self.x)) in  
obj 'inc (); obj 'inc ();  
def extobj = seal (  
    obj &  
    ('dbl _ & 'self self ->  
      self.x = self.x + self.x in self.x)) in  
extobj 'dbl (); extobj 'inc ()

    x = 1
```
Resealing Objects

```
...                   
def obj = seal (       
  'x 0 &             
  ('inc _ & 'self self ->
    self.x = self.x + 1 in self.x)) in
obj 'inc (); obj 'inc ();
def extobj = seal (   
  obj &             
  ('dbl _ & 'self self ->
    self.x = self.x + self.x in self.x)) in
extobj 'dbl (); extobj 'inc ()
```

\[ x = 2 \]
... 

```
def obj = seal ( 
  `x 0 &
  (`inc _ & `self self ->
    self.x = self.x + 1 in self.x)) in
obj `inc (); obj `inc ();
def extobj = seal ( 
  obj &
  (`dbl _ & `self self ->
    self.x = self.x + self.x in self.x)) in
extobj `dbl (); extobj `inc ()
```

\[ x = 4 \]
Resealing Objects

```python
... 
def obj = seal ( 
    'x 0 &
    ('inc _ & 'self self ->
        self.x = self.x + 1 in self.x)) in obj 'inc (); obj 'inc ();
def extobj = seal ( 
    obj &
    ('dbl _ & 'self self ->
        self.x = self.x + self.x in self.x)) in extobj 'dbl (); extobj 'inc ()

x = 5
```
Resealing Objects

... 

def obj = seal (...) in

obj `inc (); obj `inc ();
def extobj = seal (...) in

extobj `dbl (); extobj `inc ()
Resealing Objects

```
... 
2 def obj = seal (...) in
3 obj `inc (); obj `inc ();
4 def extobj = seal (...) in
5 extobj `dbl (); extobj `inc ()

`inc ()
```
Resealing Objects

```
... 

def obj = seal (...) in
obj ‘inc (); obj ‘inc ();
def extobj = seal (...) in
extobj ‘dbl (); extobj ‘inc ()

‘self extobj & ‘inc ()
```
...  

2 def obj = seal (...) in
3 obj `inc (); obj `inc ();
4 def extobj = seal (...) in
5 extobj `dbl (); extobj `inc ()

`self obj & `self extobj & `inc ()
Resealing Objects

... 

\begin{verbatim}
def obj = seal (...) in
  obj 'inc (); obj 'inc ();
def extobj = seal (...) in
  extobj 'dbl (); extobj 'inc ()

'self obj & 'self extobj & 'inc ()
\end{verbatim}
def point = seal (‘x 0 & ‘y 0 &
    (‘l1 _ & ‘self self ->
        self.x + self.y)) in
def mixin = (‘nearZero _ & ‘self self ->
    (self ‘l1 ()) <= 4) in
def mixedPoint = seal (point & mixin) in
mixedPoint ‘nearZero ()

• Mixins
Other Features

1. `def point = ... in`
2. `def mixin = ((nearZero _ & self self ->
   (self 'l1 () <= 4)) in`
3. `def mixedPoint = seal (point & mixin) in`
4. `mixedPoint 'nearZero ()`

- Mixins
- Higher-order object extension
Other Features

```python
def obj = seal (‘x 0 & (‘inc _ & ‘self self ->
    self.x = self.x + 1 in self.x)) in
def obj2 = seal (obj &. ‘x) & ‘y 0 &
    (‘inc _ & ‘self self ->
    self.y = self.y + self.x in self.y)) in
...
```

- Mixins
- Higher-order object extension
- Data sharing
Other Features

```python
def obj = seal (‘x 0 & (‘inc n:int & ‘self self ->
    self.x = self.x + n in self.x) &
  (‘inc n:unit & ‘self self ->
    self ‘inc 1) in
obj (‘inc ()); obj (‘inc 4)
```

- Mixins
- Higher-order object extension
- Data sharing
- Overloading
Other Features

etc.

- Mixins
- Higher-order object extension
- Data sharing
- Overloading
- Classes, inheritance, etc.
Type Inference
Type Inference

- Subtype constraint system
Type Inference

- Subtype constraint system
- Assign each subexpression a type variable
Type Inference

- Subtype constraint system
- Assign each subexpression a type variable
- Derive initial constraint set over expression
Type Inference

- Subtype constraint system
- Assign each subexpression a type variable
- Derive initial constraint set over expression
- Perform knowledge closure on constraints
Type Inference

- Subtype constraint system
- Assign each subexpression a type variable
- Derive initial constraint set over expression
- Perform knowledge closure on constraints
- Check resulting closure for consistency
Type Inference

- Subtype constraint system
- Assign each subexpression a type variable
- Derive initial constraint set over expression
- Perform knowledge closure on constraints
- Check resulting closure for consistency
- **Soundness** is proven over inference system
Constraint Types

\[ \text{int} \cup \text{unit} \]
Constraint Types

\[
\text{int} \cup \text{unit} \iff \alpha \setminus \{\text{int} <: \alpha, \text{unit} <: \alpha\}
\]
Constraint Closure

5 + 3
Constraint Closure

\[ 5 + 3 \]

\( \alpha_1 \)

\( \alpha_2 \)
Constraint Closure

5 + 3

\( \alpha_1 \)

\( \alpha_2 \)

\( \alpha_3 \)
Constraint Closure

\[ 5 + 3 \]

\[ \text{int} \ 
\alpha_1 \\
\alpha_2 \\
\alpha_3 \]
5 + 3
Constraint Closure

\[
\text{int} \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow + \rightarrow \alpha_3
\]

\[5 + 3\]
Constraint Closure

5 + 3
Constraint Closure

\[ 5 + 3 \]
Constraint Closure

$5 + 3$

Diagram:

- **int**
- $\alpha_1$
- $\alpha_2$
- $\alpha_3$
- **+**
Constraint Closure

\[ 5 + 3 \]
Constraint Closure

\[
\begin{align*}
\text{int} & \Rightarrow \alpha_1 \\
\alpha_1 & \Rightarrow \alpha_2 \\
\alpha_2 & \Rightarrow + \\
+ & \Rightarrow \alpha_3
\end{align*}
\]

\[5 + 3\]
Constraint Closure

\[ 5 + 3 \]
Functions

\[ x \rightarrow x + x \]
Functions

\[ x \rightarrow x + x \]
Functions

\[ x \rightarrow x + x \]
Functions

\[ x \rightarrow x + x \]
Functions

\[ x \rightarrow x + x \]
Functions

\[ x \rightarrow x + x \]
Application

\[(x \to x + x)\ 5\]
Application

$$(x \rightarrow x + x)\ 5$$
Application

\[(x \rightarrow x + x)^5\]
Application

\[(x \rightarrow x + x)^5\]
Application

\[(x \rightarrow x + x) \ 5\]
Application

\[(x \to x + x) \, 5\]
Application

\[(x \rightarrow x + x) \ 5\]
Application

\((x \rightarrow x + x) \, 5\)
Application

\[(x \rightarrow x + x) \ 5\]
Application

\[(x \rightarrow x + x) \ 5\]
Application

\((x \rightarrow x + x) \; 5\)
Application

\[(x \rightarrow x + x) \ 5\]
Application

\[(x \rightarrow x + x) \ 5\]
Application

\[(x \rightarrow x + x) \ 5\]
Application

\[(x \rightarrow x + x) \ 5 : \ \text{int}\]
Polymorphism
Polymorphism

- Let-bound polymorphism
Polymorphism

- Let-bound polymorphism
  - Type-parametric methods fail
Polymorphism

- Let-bound polymorphism
  - Type-parametric methods fail
- Local polymorphism
Polymorphism

- Let-bound polymorphism
  - Type-parametric methods fail
- Local polymorphism
  - Objects are not local
Polymorphism

- Let-bound polymorphism
  - Type-parametric methods fail
- Local polymorphism
  - Objects are not local
  - Requires type annotations
Polymorphism

- Let-bound polymorphism
  - Type-parametric methods fail
- Local polymorphism
  - Objects are not local
  - Requires type annotations
- TinyBang uses call-site polymorphism
Let-bound polymorphism
- Type-parametric methods fail

Local polymorphism
- Objects are not local
- Requires type annotations

TinyBang uses call-site polymorphism
- Each call site is freshly polyinstantiated
Let-bound polymorphism
- Type-parametric methods fail

Local polymorphism
- Objects are not local
- Requires type annotations

TinyBang uses call-site polymorphism
- Each call site is freshly polyinstantiated
- Recursion reuses variable contours
Polymorphic Application

def id = x → x in (id () & id 1)
def id = x -> x in (id () & id 1)
Polymorphic Application

```plaintext
def id = x -> x in (id () & id 1)
```
Polymorphic Application

```
def id = x -> x in (id () & id 1)
```
def id = x -> x in (id () & id 1)
BigBang

- Aims to infer types for script-like programs
BigBang

- Aims to infer types for script-like programs
- Uses type information for better performance
BigBang

- Aims to infer types for script-like programs
- Uses type information for better performance
- Desugars down to TinyBang
BigBang

- Aims to infer types for script-like programs
- Uses type information for better performance
- Desugars down to TinyBang
- Provides syntax for classes, modules, etc.
BigBang

- Aims to infer types for script-like programs
- Uses type information for better performance
- Desugars down to TinyBang
- Provides syntax for classes, modules, etc.
- Enough polymorphism for scripting intuitions
BigBang

- Aims to infer types for script-like programs
- Uses type information for better performance
- Desugars down to TinyBang
- Provides syntax for classes, modules, etc.
- Enough polymorphism for scripting intuitions
- ...without divergence or exponential blow-up
TinyBang encodes objects as scapes and onions
Summary

- TinyBang encodes objects as scapes and onions
- Variant destruction is heterogeneously typed
Summary

- TinyBang encodes objects as scapes and onions
- Variant destruction is heterogeneously typed
- (Re)sealing is encodable as a function
Summary

- TinyBang encodes objects as scapes and onions
- Variant destruction is heterogeneously typed
- (Re)sealing is encodable as a function
- Flexible OO structures are encodable
Summary

- TinyBang encodes objects as scapes and onions
- Variant destruction is heterogeneously typed
- (Re)sealing is encodable as a function
- Flexible OO structures are encodable
- Heavily uses call-site polymorphism model
Summary

- TinyBang encodes objects as scapes and onions
- Variant destruction is heterogeneously typed
- (Re)sealing is encodable as a function
- Flexible OO structures are encodable
- Heavily uses call-site polymorphism model
- Requires whole-program typechecking
Summary

- TinyBang encodes objects as scapes and onions
- Variant destruction is heterogeneously typed
- (Re)sealing is encodable as a function
- Flexible OO structures are encodable
- Heavily uses call-site polymorphism model
- Requires whole-program typechecking

Questions?