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

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)

```
object a {
}
```

(OCaml)

```
let a = {
}
```

Record-Based Object Encoding

(Scala)

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

(OCaml)

```ocaml
let a = {
  v = ref 5
}
```

- Object fields are record fields

Record-Based Object Encoding

(Scala)

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

(OCaml)

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

- Object fields are record fields
- Methods are fields with functions

Record-Based Object Encoding

(Scala)

```
object a {
 val v = 5
 def mth(x:Int)
   :Int = { x+v }
 def foo(x:Unit){}
};
a.mth(3)
```

(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

Record-Based Object Encoding

(Scala)

```
object a {
 val v = 5
 def mth(x:Int)
   :Int = { x+v }
 def foo(x:Unit){}
};
a.mth(3)
```

(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 $\Longleftrightarrow$ 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)

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

(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 message handling cases

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 message handling cases
- 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 message handling cases
- Invocation is just message passing
- But this doesn't typecheck!

Typing Variant Destruction

```
match v with
  | `Odd y -> y mod 2 = 1
  | `Dbl x -> x + x
```

- Typechecking variant destruction is tricky

Typing Variant Destruction

```
match v with
   | `Odd y -> y mod 2 = 1
   | `Dbl x -> x + x
```

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

Typing Variant Destruction

```
match v with
  | `Odd y -> y mod 2 = 1
  | `Dbl x -> x + x
```

$: \texttt{int} \cup \texttt{bool}$

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

Typing Variant Destruction

```
match `Dbl 2 with
  | `Odd y -> y mod 2 = 1
  | `Dbl x -> x + x
```

: int ∪ bool

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

Typing Variant Destruction

```
match `Dbl 2 with
  | `Odd y -> y mod 2 = 1   : int!
  | `Dbl x -> 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

```
match `Dbl 2 with
  | `Odd y -> y mod 2 = 1  : int!
  | `Dbl x -> 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)

+

χ ->

Scapes

(Functions with built-in patterns)

Variant-Based Object Encoding

TinyBang

```
`dbl x -> x + x
```

- Methods are scapes

Variant-Based Object Encoding

TinyBang

```
`dbl x -> x + x
```

- Methods are scapes: functions with patterns

Variant-Based Object Encoding

TinyBang

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

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

Variant-Based Object Encoding

TinyBang

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

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

Variant-Based Object Encoding

TinyBang

```
(`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

```
‘dbl x -> x + x
```

Many Methods: Onioning Scapes

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

- Scapes are combined by *onioning*

Many Methods: Onioning Scapes

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

- Scapes are combined by *onioning*
- Application finds match

Many Methods: Onioning Scapes

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

```
object a {
  def dbl(x:Int):Int = { x + x }
  def pos(y:Int):Boolean = { y % 2 == 1 }
}
a.dbl(2)
```

Many Methods: Onioning Scapes

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

$\Rightarrow 4$

- Scapes are combined by *onioning*
- Application finds rightmost match (asymmetric)

Many Methods: Onioning Scapes

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

$$\Rightarrow \; 4$$

- Scapes are combined by *onioning*
- Application finds rightmost match (asymmetric)
- Subsumes case expressions

Many Methods: Onioning Scapes

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

$\Longrightarrow 4$

- Scapes are combined by *onioning*
- Application finds rightmost match (asymmetric)
- Subsumes case expressions
- Generalizes First-Class Cases [Blume et. al. '06]

Typing the Onion

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

(`dbl int ∪ `odd int) -> (int ∪ bool)

- Simple union type loses alignment

Typing the Onion

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

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

- Simple union type loses alignment
- *Onion type* does not

Typing the Onion

```
(‘dbl x -> x + x) &
(‘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

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

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

- Simple union type loses alignment
- *Onion type* does not
- Weakly dependent type
- Relies heavily on polymorphism

Fields

- Pure variant model: get/set messages

Fields

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

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

Fields

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

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

Fields

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

- ~~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

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

```
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

```
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
- But what about self?

Naïve Self

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

Naïve Self

```
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

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

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

Naïve Self

```
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

```
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

```
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

$\alpha_{\text{SELF}} =$

```
(`foo _ & `self α1 -> int) &
(`bar _ -> int)
    where α1 has `bar
```

$$\bigcup$$

```
(`foo _ & `self α2 -> int) &
(`baz _ -> int)
    where α2 has `baz
```

Naïve Self: Problems

$$\alpha_{\text{SELF}} \quad :> \quad \begin{array}{l} \texttt{(`foo \_ \& `self } \alpha_1 \texttt{ -> int) \&} \\ \texttt{(`bar \_ -> int)} \\ \qquad \texttt{where } \alpha_1 \texttt{ has `bar} \end{array}$$

$$\alpha_{\text{SELF}} \quad :> \quad \begin{array}{l} \texttt{(`foo \_ \& `self } \alpha_2 \texttt{ -> int) \&} \\ \texttt{(`baz \_ -> int)} \\ \qquad \texttt{where } \alpha_2 \texttt{ has `baz} \end{array}$$

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**

Sealing in TinyBang

```
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 ()
```

$$x = 2$$

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 ()
```

$$x = 4$$

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 ()
```

$$x = 5$$

Resealing Objects

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

Resealing Objects

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

Resealing Objects

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

```
'self obj & 'self extobj & 'inc ()
```

Resealing Objects

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

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

Other Features

```
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

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

- Mixins
- Higher-order object extension

Other Features

```
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

```
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

$\textbf{int} \cup \textbf{unit}$

Constraint Types

$$\textbf{int} \cup \textbf{unit}$$

$$\Updownarrow$$

$$\alpha \backslash \{\textbf{int} <: \alpha, \textbf{unit} <: \alpha\}$$

Constraint Closure

5 + 3

Constraint Closure

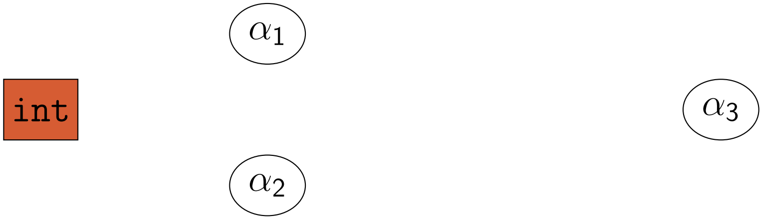
5 + 3

Constraint Closure

5 + 3

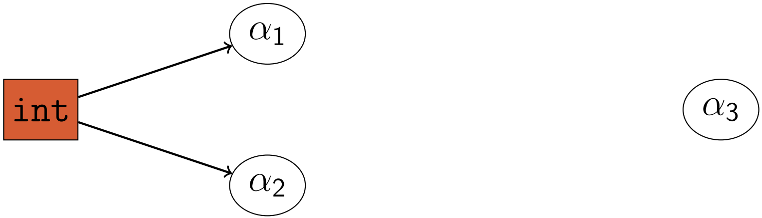
Constraint Closure

`5 + 3`



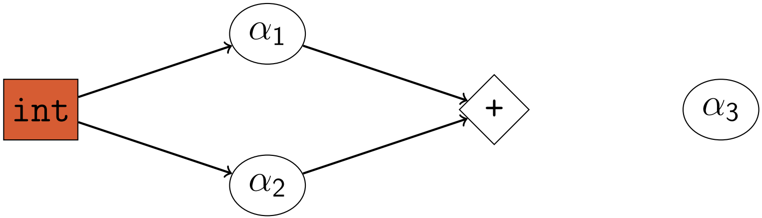
Constraint Closure

5 + 3



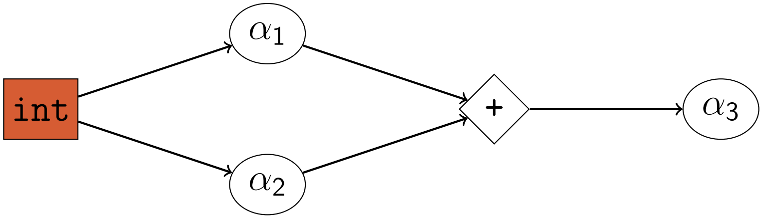
Constraint Closure

`5 + 3`



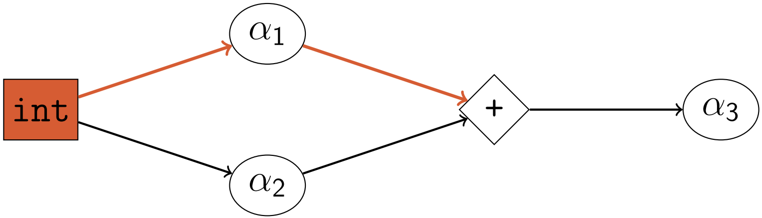
Constraint Closure

`5 + 3`



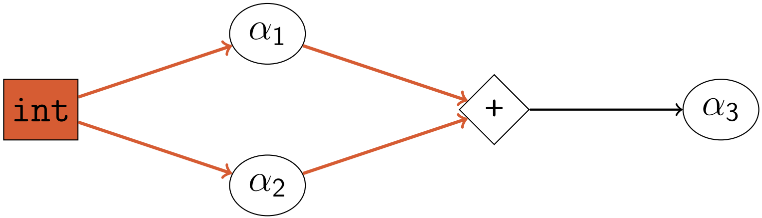
Constraint Closure

`5 + 3`



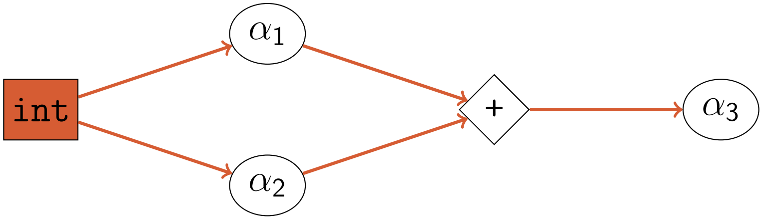
Constraint Closure

`5 + 3`



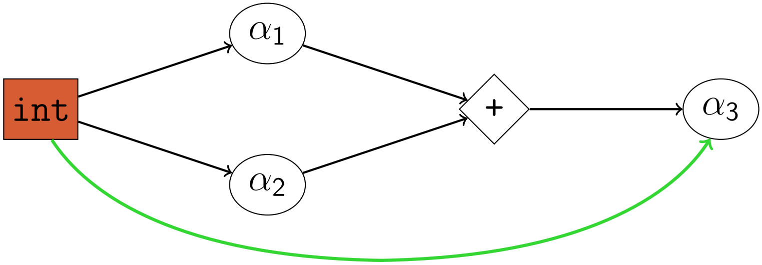
Constraint Closure

`5 + 3`



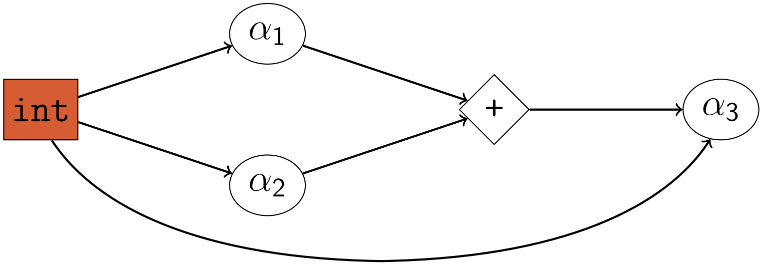
Constraint Closure

`5 + 3`



Constraint Closure

`5 + 3`

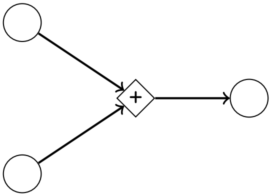


Functions

```
x -> x + x
```

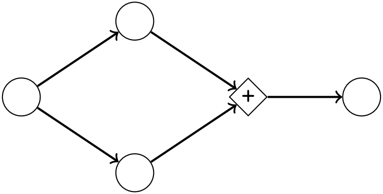
Functions

`x -> x + x`



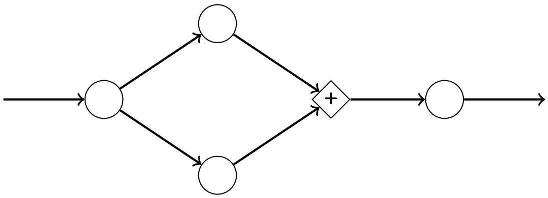
Functions

```
x -> x + x
```



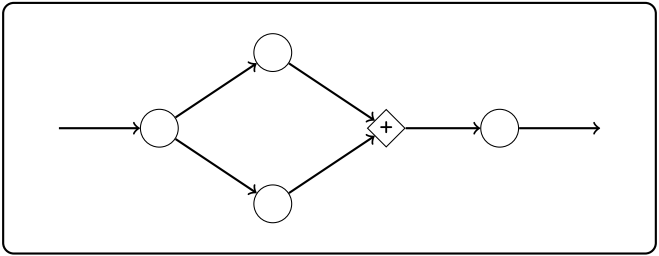
Functions

`x -> x + x`



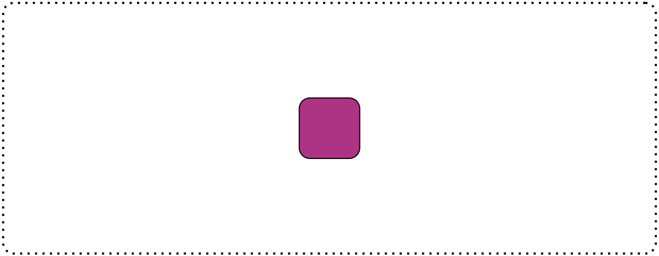
Functions

```
x -> x + x
```



Functions

```
x -> x + x
```



Application

```
(x -> x + x) 5
```

Application

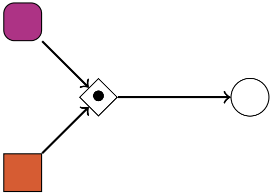
`(x -> x + x) 5`

Application

```
(x -> x + x) 5
```

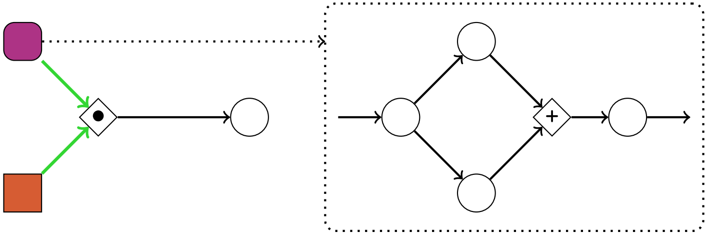
Application

`(x -> x + x) 5`



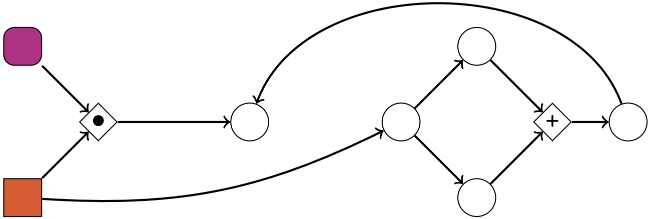
Application

`(x -> x + x) 5`



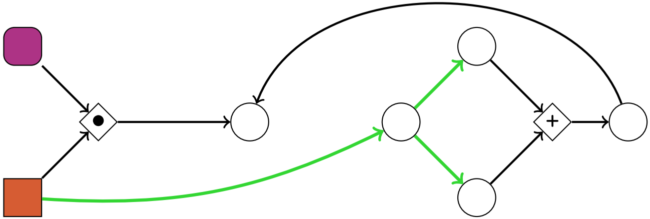
Application

`(x -> x + x) 5`



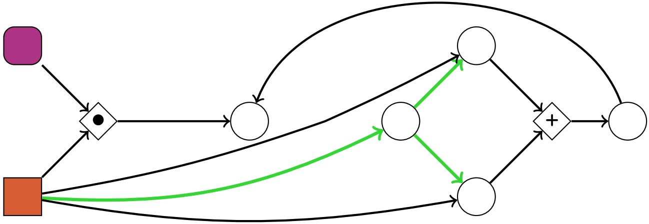
Application

```
(x -> x + x) 5
```



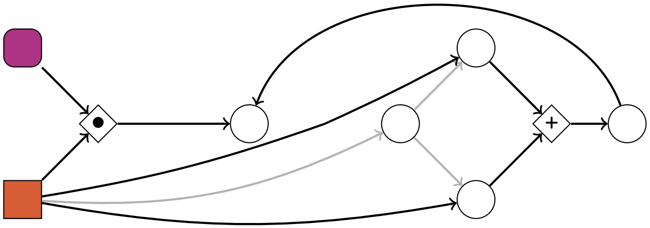
Application

`(x -> x + x) 5`



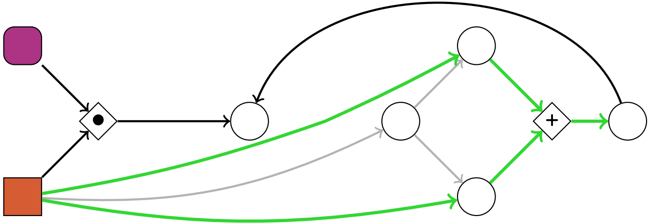
Application

```
(x -> x + x) 5
```



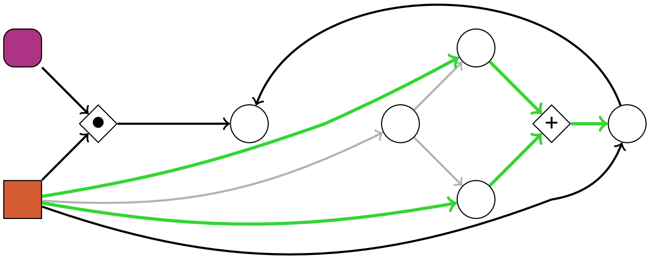
Application

```
(x -> x + x) 5
```



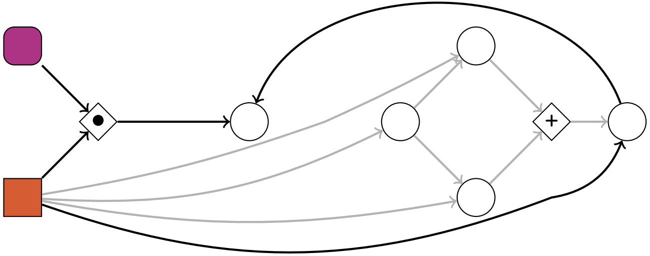
Application

`(x -> x + x) 5`



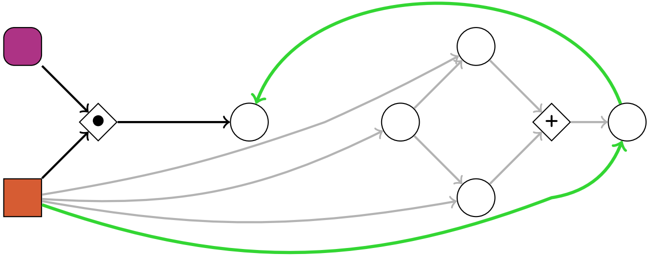
Application

```
(x -> x + x) 5
```



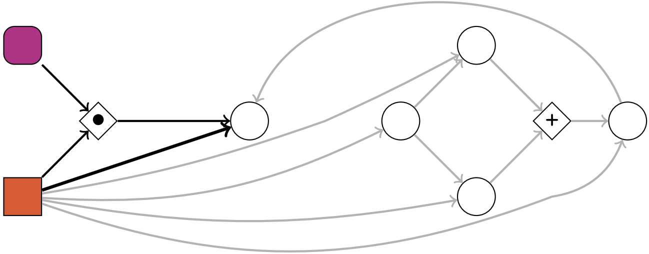
Application

```
(x -> x + x) 5
```



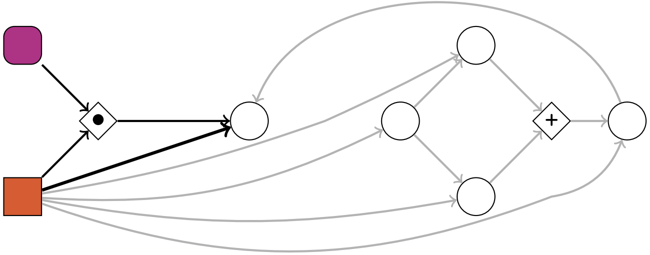
Application

`(x -> x + x) 5`



Application

`(x -> x + x) 5 :  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

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

Polymorphism

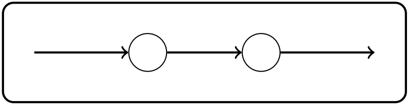
- Let-bound polymorphism
  - Type-parametric methods fail
- Local polymorphism
  - Objects are not local
  - Requires type annotations
- TinyBang uses call-site polymorphism
  - Each call site is freshly polyinstantiated
  - Recursion reuses variable contours

Polymorphic Application

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

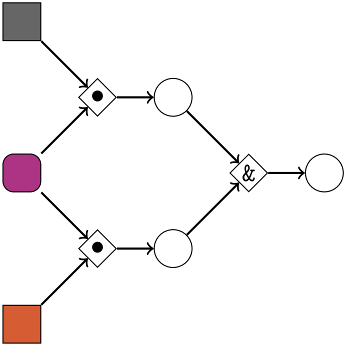
Polymorphic Application

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



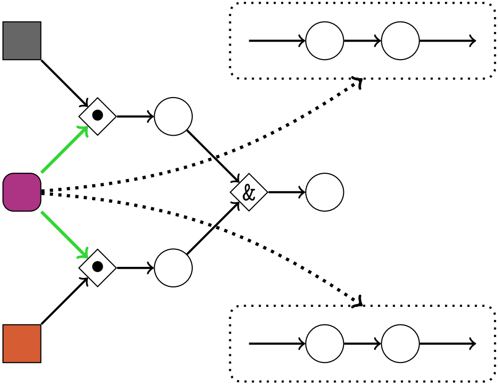
Polymorphic Application

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



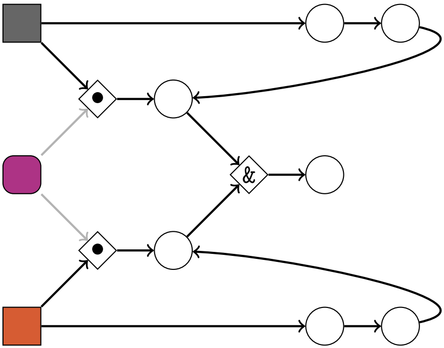
Polymorphic Application

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



Polymorphic Application

```
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

Summary

- 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?