Types for Flexible Objects
Building a Typed Scripting Language

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

The Johns Hopkins University

November 17th, 2014
Objective

Flexible OO language
Objective

Flexible OO language

+ Static typing, inference, etc.
What is “flexible?”

- Freely transition between views on data
What is “flexible?”

- Freely transition between views on data
- Freely manipulate data at runtime
What is “flexible?”

- Freely transition between views on data
- Freely manipulate data at runtime
- Python, Javascript, Ruby
What is “flexible?”

- Freely transition between views on data
- Freely manipulate data at runtime
- Python, Javascript, Ruby
  - Objects are dictionaries (almost seamlessly)
What is “flexible?”

- Freely transition between views on data
- Freely manipulate data at runtime
- Python, Javascript, Ruby
  - Objects are dictionaries (almost seamlessly)
  - Dynamic lookup, update, etc. based on runtime conditions
What is “flexible?”

- Freely transition between views on data
- Freely manipulate data at runtime
- Python, Javascript, Ruby
  - Objects are dictionaries (almost seamlessly)
  - Dynamic lookup, update, etc. based on runtime conditions
- Minimal semantic clutter (for readability)
What is “flexible?”

- Freely transition between views on data
- Freely manipulate data at runtime
- Python, Javascript, Ruby
  - Objects are dictionaries (almost seamlessly)
  - Dynamic lookup, update, etc. based on runtime conditions
- Minimal semantic clutter (for readability)
- Good for ad-hoc, situational requirements
Examples of Flexibility

Add a method to an existing object:

class A:
    def foo(): return 1
a = A()
a.bar = types.MethodType(
    lambda self: 2, a)
Examples of Flexibility

Rely on execution path invariants:

```python
def neg(x):
    if type(x) is int:
        return -x
    else:
        return not x

print neg(2) + 3
```
Why Types?

- Flexible languages are good:

  - Faster development
  - Capture complex ideas with clever patterns
  - Structurally typed ("duck typing")

Static types are good:

  - Describe invariants on data
  - Help programmer understanding
  - Statically identify programming errors
  - Improve runtime performance
  - Other static invariants: immutability, limiting data scope, etc.

We want both!
Why Types?

- Flexible languages are good:
  - Faster development

- Static types are good:
  - Describe invariants on data
  - Help programmer understanding
  - Statically identify programming errors
  - Improve runtime performance
  - Other static invariants: immutability, limiting data scope, etc.

We want both!
Why Types?

Flexible languages are good:
- Faster development
- Capture complex ideas with clever patterns

Static types are good:
- Describe invariants on data
- Help programmer understanding
- Statically identify programming errors
- Improve runtime performance
- Other static invariants: immutability, limiting data scope, etc.

We want both!
Why Types?

Flexible languages are good:
- Faster development
- Capture complex ideas with clever patterns
- Structurally typed ("duck typing")

Static types are good:
- Describe invariants on data
- Help programmer understanding
- Statically identify programming errors
- Improve runtime performance
- Other static invariants: immutability, limiting data scope, etc.

We want both!
Why Types?

- Flexible languages are good:
  - Faster development
  - Capture complex ideas with clever patterns
  - Structurally typed ("duck typing")

- Static types are good:
Why Types?

- Flexible languages are good:
  - Faster development
  - Capture complex ideas with clever patterns
  - Structurally typed ("duck typing")

- Static types are good:
  - Describe invariants on data
Why Types?

- Flexible languages are good:
  - Faster development
  - Capture complex ideas with clever patterns
  - Structurally typed ("duck typing")

- Static types are good:
  - Describe invariants on data
  - Help programmer understanding
Why Types?

Flexible languages are good:
- Faster development
- Capture complex ideas with clever patterns
- Structurally typed ("duck typing")

Static types are good:
- Describe invariants on data
- Help programmer understanding
- Statically identify programming errors
Why Types?

- Flexible languages are good:
  - Faster development
  - Capture complex ideas with clever patterns
  - Structurally typed ("duck typing")

- Static types are good:
  - Describe invariants on data
  - Help programmer understanding
  - Statically identify programming errors
  - Improve runtime performance
Why Types?

- Flexible languages are good:
  - Faster development
  - Capture complex ideas with clever patterns
  - Structurally typed (“duck typing”)

- Static types are good:
  - Describe invariants on data
  - Help programmer understanding
  - Statically identify programming errors
  - Improve runtime performance
  - Other static invariants: immutability, limiting data scope, etc.
Why Types?

- Flexible languages are good:
  - Faster development
  - Capture complex ideas with clever patterns
  - Structurally typed (“duck typing”)

- Static types are good:
  - Describe invariants on data
  - Help programmer understanding
  - Statically identify programming errors
  - Improve runtime performance
  - Other static invariants: immutability, limiting data scope, etc.

- We want both!
Options?

- Powerful traditional type system
  - 😊 Captures static invariants (monads, dep. types)
  - 😞 Often incurs semantic clutter
  - 😞 High barrier to entry
  - *Not enough flexibility*

Build type inference system for existing language (e.g. DRuby)

Fundamentally flexible language

Limited success: language features defy static typing (e.g. mutable monkeypatching, Python locals)

Not enough static typing

Design a language to include statically-typed flex
Options?

- Powerful traditional type system
  - 😊 Captures static invariants (monads, dep. types)
  - 😞 Often incurs semantic clutter
  - 😞 High barrier to entry
  - *Not enough flexibility*

- Build type inference system for existing language (e.g. DRuby)
  - 😊 Fundamentally flexible language
  - 😞 Limited success: language features defy static typing (e.g. mutable monkeypatching, Python locals)
  - *Not enough static typing*
Options?

- Powerful traditional type system
  - 🌟 Captures static invariants (monads, dep. types)
  - 😞 Often incurs semantic clutter
  - 😞 High barrier to entry
  - Not enough flexibility

- Build type inference system for existing language (e.g. DRuby)
  - 🌟 Fundamentally flexible language
  - 😞 Limited success: language features defy static typing (e.g. mutable monkeypatching, Python locals)
  - Not enough static typing

- Design a language to include statically-typed flex
TinyBang

- Start with an ML-like basis
TinyBang

- Start with an ML-like basis
  - records, variants, patterns, let, refs, inference
TinyBang

- Start with an ML-like basis
  - records, variants, patterns, let, refs, inference

- Add flexible properties
  - All types inferred – no declarations
  - Structural subtyping ("duck typing")
  - Powerful record combinators
  - First-class match clauses
  - Refinement on pattern matching

Result: flexible language with static typing
TinyBang

- Start with an ML-like basis
  - records, variants, patterns, let, refs, inference
- Add flexible properties
  - All types inferred – no declarations
  - *Structural* subtyping (“duck typing”)
  - Powerful record combinators
  - First-class match clauses
  - Refinement on pattern matching
- **Result:** flexible language with static typing
Outline

- Semantics
  - Powerful records and record combinators
  - First-class match clauses
  - Object encoding using variants

- Static typing
  - Overview
  - Union elimination
  - Polymorphism

- Summary
Asymmetric Concatenation

- Use type-indexed records [Blume et. al. ’06]
Asymmetric Concatenation

- Use *type-indexed* records [Blume et. al. ’06]
  - \{\text{int} = 4, \text{A} = \{\text{int} = 5\}\}

Note: 1-ary record = 1-ary variant

Concatenation asymmetrically prefers left components

Asymmetry is fundamental in several encodings
Asymmetric Concatenation

- Use *type-indexed* records [Blume et al. ’06]
  - \{\text{int} = 4, \ A = \{\text{int} = 5\}\}
  - 4 & ‘A 5
Asymmetric Concatenation

- Use *type-indexed* records [Blume et. al. ’06]
  - \{\text{int} = 4, \text{A} = \{\text{int} = 5\}\}
  - 4 \& ‘A 5
- Note: 1-ary record = 1-ary variant
Asymmetric Concatenation

- Use *type-indexed* records [Blume et al. ’06]
  - \{\text{int} = 4, \ A = \{\text{int} = 5\}\}
  - \(4 \& \ A\ 5\)
  - Note: 1-ary record = 1-ary variant

- Concatenation asymmetrically prefers left components
  - \(\sim\)
  - \(\sim\)
  - \(\sim\)
  - \(\sim\)
  - \(\sim\)
  - \(\sim\)

\[
\begin{align*}
\text{Asymmetry is fundamental in several encodings}
\end{align*}
\]
Asymmetric Concatenation

- Use *type-indexed* records [Blume et. al. ’06]
  - \{\text{int} = 4, A = \{\text{int} = 5\}\}
  - $4 \& 'A 5$
  - Note: 1-ary record = 1-ary variant

- Concatenation asymmetrically prefers left components
  - $'A 4 \& 'B 6 \& 'A 3 \simeq 'A 4 \& 'B 6$

- Asymmetry is fundamental in several encodings
Outline

- Semantics
  - Powerful records and record combinators
  - First-class match clauses
  - Object encoding using variants

- Static typing
  - Overview
  - Union elimination
  - Polymorphism

- Summary
Partial Functions

All TinyBang functions are *partial* via pattern-matching:
Partial Functions

All TinyBang functions are *partial* via pattern-matching:

- `int -> 0` matches all records with an `int` component
Partial Functions

All TinyBang functions are *partial* via pattern-matching:

- `int -> 0` matches all records with an `int` component
- `int & x -> x` is integer identity
  - In patterns, `&` is conjunction
  - Variables (e.g. `x`) match anything and bind it
Partial Functions

All TinyBang functions are partial via pattern-matching:

- **int** -> 0 matches all records with an int component
- **int & x** -> x is integer identity
  - In patterns, & is conjunction
  - Variables (e.g. x) match anything and bind it

Pattern-matching takes the leftmost match

- (‘B x -> x+1) (‘A 3 & ‘B 1) ⇒ 2
- (‘B x -> x+1) (‘B 5 & ‘A 3 & ‘B 1) ⇒ 6
Partial Functions

All TinyBang functions are partial via pattern-matching:

- **int** -> 0 matches all records with an **int** component
- **int & x -> x** is integer identity
  - In patterns, **&** is conjunction
  - Variables (e.g. **x**) match anything and bind it
- Pattern-matching takes the leftmost match
  - (**'B x -> x+1**) (**'A 3 & 'B 1**) ⇒ 2
  - (**'B x -> x+1**) (**'B 5 & 'A 3 & 'B 1**) ⇒ 6
Extensible match clauses

- Single clause: write as function
  \[ \langle 'l \, x \, \& \, 'r \, y \, \rightarrow \, x \, + \, y \rangle \]
Extensible match clauses

- Single clause: write as function
  \[
  (\text{`l } x \& \text{ `r } y \rightarrow x + y)
  \]

- Concatenate functions (\&) for multiple clauses
  \[
  (\text{`l } x \& \text{ `r } y \rightarrow x + y) \&
  (\text{int } \& \text{ z } \rightarrow z + 1)
  \]
Extensible match clauses

- Single clause: write as function
  \[ (\text{'l } x \text{ & } \text{'r } y \rightarrow x + y) \]

- Concatenate functions (\&) for multiple clauses
  \[ (\text{'l } x \text{ & } \text{'r } y \rightarrow x + y) \& (\text{int } \& z \rightarrow z + 1) \]

- Encodes match!
Extensible match clauses

- Single clause: write as function
  \[(l \ x \ & \ r \ y \rightarrow x + y)\]

- Concatenate functions (&) for multiple clauses
  \[(l \ x \ & \ r \ y \rightarrow x + y) \& (\text{int} \ & \ z \rightarrow z + 1)\]

- Encodes match!

- Operator & uniformly concatenates records, match clauses: \[('x \ 1) \& (\text{int} \rightarrow 0)\]
Outline

- Semantics
  - Powerful records and record combinators
  - First-class match clauses
  - Object encoding using variants

- Static typing
  - Overview
  - Union elimination
  - Polymorphism

- Summary
Variant-Based Object Encoding

```
let rec seal = (see paper) in
let prePoint =
  'x 2 & 'y 4 &
  ('mg & 'self slf -> slf 'gx + slf 'gy) &
  ('gx & 'self slf -> slf . x) & ...
in let point = seal prePoint in
point 'mg // returns 6

& can combine data and match clauses
```
Variant-Based Object Encoding

```haskell
let rec seal = (see paper) in
let prePoint =
  'x 2 & 'y 4 &
  ('mg & 'self slf -> slf 'gx + slf 'gy ) &
  ('gx & 'self slf -> slf .x) & ...
in let point = seal prePoint in
point 'mg  // returns 6

• & can combine data and match clauses
• Match clause encoding of objects, not records
```
Variant-Based Object Encoding

```ml
let rec seal = (see paper) in
let prePoint =
  'x 2 & 'y 4 &
  ('mg & 'self slf -> slf 'gx + slf 'gy ) &
  ('gx & 'self slf -> slf.x) & ...
in let point = seal prePoint in
point 'mg  // returns 6
```

- & can combine data and match clauses
- *Match clause* encoding of objects, not records
- Seal is a combinator tying self-reference knot
Variant-Based Object Encoding

```
let rec seal = (see paper) in
let prePoint =
  'x 2 & 'y 4 &
  ('mg & 'self slf -> slf 'gx + slf 'gy ) &
  ('gx & 'self slf -> slf.x) & ... in
let point = seal prePoint in
point 'mg  // returns 6
```

- & can combine data and match clauses
- *Match clause* encoding of objects, not records
- Seal is a combinator tying self-reference knot
- Can seal, message, extend, reseal, message
Variant-Based Object Encoding

```ocaml
let rec seal = (see paper) in
let prePoint =
  `x 2 & `y 4 &
  (‘mg & ‘self slf -> slf ‘gx + slf ‘gy ) &
  (‘gx & ‘self slf -> slf.x) & ...
in let point = seal prePoint in
point ‘mg // returns 6
```

- & can combine data and match clauses
- *Match clause* encoding of objects, not records
- Seal is a combinator tying self-reference knot
- Can seal, message, extend, reseal, message
- All statically typed!
Other Encodings

Together, type-indexed records and partial functions can encode:

- Conditionals (\texttt{True () \text{and} False ()})
- Variant-based objects
- Operator overloading
- Classes, inheritance, subclasses, etc.
- Mixins, dynamic functional object extension
- First-class cases
- Optional arguments
- etc.

And we can type them!
Other Encodings
Together, type-indexed records and partial functions can encode:

- Conditionals (‘True () and ‘False ())
- Variant-based objects
- Operator overloading
- Classes, inheritance, subclasses, etc.
- Mixins, dynamic functional object extension
- First-class cases
- Optional arguments
- etc.
Other Encodings

Together, type-indexed records and partial functions can encode:

- Conditionals (‘True () and ‘False ()
- Variant-based objects
- Operator overloading
- Classes, inheritance, subclasses, etc.
- Mixins, dynamic functional object extension
- First-class cases
- Optional arguments
- etc.

And we can type them!
Outline

- Semantics
  - Powerful records and record combinators
  - First-class match clauses
  - Object encoding using variants

- Static typing
  - Overview
  - Union elimination
  - Polymorphism

- Summary
Types

- Rooted in subtype constraint inference
  (Aiken/Heintze/Pottier)
Types

- Rooted in subtype constraint inference
  (Aiken/Heintze/Pottier)
- With new notions of

Union elimination: path sensitivity
Polymorphism: context sensitivity
Concession: it's whole-program typing
Types

- Rooted in subtype constraint inference
  (Aiken/Heintze/Pottier)
- With new notions of
  - Union type elimination
- Polymorphism
- Metatheory (rule system, soundness, etc)
- Inspired by program analysis
- Concession: it's whole-program typing
Types

- Rooted in subtype constraint inference (Aiken/Heintze/Pottier)
- With new notions of
  - Union type elimination
  - Polymorphism

Inspired by program analysis.

Metatheory (rule system, soundness, etc)

Union elimination: path sensitivity
Polymorphism: context sensitivity

Concession: it's whole-program typing
Types

- Rooted in subtype constraint inference
  (Aiken/Heintze/Pottier)
- With new notions of
  - Union type elimination
  - Polymorphism
  - Metatheory (rule system, soundness, etc)
Types

- Rooted in subtype constraint inference
  (Aiken/Heintze/Pottier)
- With new notions of
  - Union type elimination
  - Polymorphism
  - Metatheory (rule system, soundness, etc)
- Inspired by program analysis
Types

- Rooted in subtype constraint inference (Aiken/Heintze/Pottier)
- With new notions of
  - Union type elimination
  - Polymorphism
  - Metatheory (rule system, soundness, etc)
- Inspired by program analysis
  - Union elimination: path sensitivity
Types

- Rooted in subtype constraint inference (Aiken/Heintze/Pottier)
- With new notions of
  - Union type elimination
  - Polymorphism
  - Metatheatery (rule system, soundness, etc)
- Inspired by program analysis
  - Union elimination: path sensitivity
  - Polymorphism: context sensitivity
Types

- Rooted in subtype constraint inference
  (Aiken/Heintze/Pottier)
- With new notions of
  - Union type elimination
  - Polymorphism
  - Metatheory (rule system, soundness, etc)
- Inspired by program analysis
  - Union elimination: path sensitivity
  - Polymorphism: context sensitivity
  - Concession: it’s whole-program typing
Outline

Semantics
- Powerful records and record combinators
- First-class match clauses
- Object encoding using variants

Static typing
- Overview
- Union elimination
- Polymorphism

Summary
Union type elimination

- Example: patterns 'z and 's 's x on Peano number input
Union type elimination

Example: patterns \( \texttt{Z} \) and \( \texttt{S} \) \( \texttt{S} \) \( \texttt{x} \) on Peano number input

\[
\alpha = \bigcup_{i} \texttt{Z} \quad \text{slice} \quad \texttt{S} \quad \text{slice} \quad \texttt{S}
\]

\[
\begin{align*}
\texttt{Z} & \rightarrow (\alpha) \\
\texttt{S} & \rightarrow (\alpha) \\
\end{align*}
\]

\[
\begin{align*}
\texttt{Z} & \rightarrow (\alpha) \\
\texttt{S} & \rightarrow (\alpha) \\
\end{align*}
\]

\[
\begin{align*}
\texttt{Z} & \rightarrow (\alpha) \\
\texttt{S} & \rightarrow (\alpha) \\
\end{align*}
\]
Union type elimination

Example: patterns 'Z and 'S 'S x on Peano number input

- Adaptively eliminate union to depth of pattern
Union type elimination

Example: patterns `Z and `S `S x on Peano number input

- Adaptively eliminate union to depth of pattern
- Slices specialize argument type
Example: patterns ‘Z and ‘S ‘S x on Peano number input

Adaptively eliminate union to depth of pattern
Slices specialize argument type

let f = (x: ‘A () -> x.B) &
  (y: ‘P () -> 1)
in f (‘A () & ‘B 5) + f ‘P ()
Outline

- **Semantics**
  - Powerful records and record combinators
  - First-class match clauses
  - Object encoding using variants

- **Static typing**
  - Overview
  - Union elimination
  - **Polymorphism**

- **Summary**
Inferred function polymorphism

- let-polymorphism ignores call context
Inferred function polymorphism

- let-polymorphism ignores call context
- Program analyses have polyvariance, e.g. $n$CFA but are brittle to refactoring
  —arbitrary cutoff at $n$ depth
Inferred function polymorphism

- let-polymorphism ignores call context
- Program analyses have polyvariance, e.g. nCFA but are brittle to refactoring
  —arbitrary cutoff at $n$ depth
- Our approach: polyvariance approach with regular expression call strings
Inferred function polymorphism

- let-polymorphism ignores call context
- Program analyses have polyvariance, e.g. \( n \text{CFA} \)
  but are brittle to refactoring
  —arbitrary cutoff at \( n \) depth
- Our approach: polyvariance approach with
  regular expression call strings
  - Limits polymorphism of recursion but nothing else
Inferred function polymorphism

- let-polymorphism ignores call context
- Program analyses have polyvariance, e.g. nCFA but are brittle to refactoring
  —arbitrary cutoff at $n$ depth
- Our approach: polyvariance approach with regular expression call strings
  - Limits polymorphism of recursion but nothing else
  - Similar to DCPA and $\Delta$CFA, but optimistic
Outline

- Semantics
  - Powerful records and record combinators
  - First-class match clauses
  - Object encoding using variants

- Static typing
  - Overview
  - Union elimination
  - Polymorphism

- Summary
Existing scripting languages are fundamentally untypeable
Summary

- Existing scripting languages are fundamentally untypeable
- So, start with ML and add "principled flex"
Summary

- Existing scripting languages are fundamentally untypeable
- So, start with ML and add “principled flex”
- TinyBang is our initial result
  —supports flex but more safely/declaratively
TinyBang is the currently implemented core
Bigger Picture: BigBang

- TinyBang is the currently implemented core
- Building a larger language: *code in it*
Bigger Picture: BigBang

- TinyBang is the currently implemented core
- Building a larger language: code in it
- Also developing compile-time dispatch methodology

http://big-bang-lang.org

Bigger Picture: BigBang

- TinyBang is the currently implemented core
- Building a larger language: code in it
- Also developing compile-time dispatch methodology
- http://big-bang-lang.org
Related Work

- Re-sealing objects generalizes [Fisher Bono ’98]
- Unifying records and variants from [Pottier ’00]
- Type-indexed records [Shields Meijer ’01]
- First-class match generalizes [Blume et. al. ’06]
- CDuce [Castagna et. al. ’14]
  - Similar expressiveness in several dimensions
  - CDuce: type checking; TinyBang: type inference
Questions?