Big Bang
Designing a Statically Typed Scripting Language

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

The Johns Hopkins University

June 11, 2012
Scripting Languages

- Terse
- Flexible
- Easy to learn
- Amenable to rapid development
Scripting Languages

- Terse
- Flexible
- Easy to learn
- Amenable to rapid development

✗ Dynamically typed
Advantages of Static Typing

- Performance
- Debugging
- Programmer understanding
Typing Existing Scripting Languages

- e.g. DRuby, Typed Racket
Typing Existing Scripting Languages

- e.g. DRuby, Typed Racket
- Existing language features are hard to type
Typing Existing Scripting Languages

- e.g. DRuby, Typed Racket
- Existing language features are hard to type
  - DRuby does not typecheck the entire Ruby API
Typing Existing Scripting Languages

- e.g. DRuby, Typed Racket
- Existing language features are hard to type
  - DRuby does not typecheck the entire Ruby API
  - Typechecking runtime metaprogramming is hard
Typing Existing Scripting Languages

- e.g. DRuby, Typed Racket
- Existing language features are hard to type
  - DRuby does not typecheck the entire Ruby API
  - Typechecking runtime metaprogramming is hard
- These systems require programmer annotation
Typing Existing Scripting Languages

- e.g. DRuby, Typed Racket
- Existing language features are hard to type
  - DRuby does not typecheck the entire Ruby API
  - Typechecking runtime metaprogramming is hard
- These systems require programmer annotation
  - Type annotations reduce terseness
Typing Existing Scripting Languages

- e.g. DRuby, Typed Racket
- Existing language features are hard to type
  - DRuby does not typecheck the entire Ruby API
  - Typechecking runtime metaprogramming is hard
- These systems require programmer annotation
  - Type annotations reduce terseness
  - Annotations can be overly restrictive
Let’s try designing a typed scripting language from scratch
Designing a Typed Scripting Language

- Design type system and execution model concurrently

- Be minimalistic: most features are encoded

- Use static near-equivalents for dynamic patterns

- Infer all types: no type declarations

- Use a whole-program typechecking model

- Use type information to improve runtime memory layout
Designing a Typed Scripting Language

- Design type system and execution model concurrently
- Be minimalistic: most features are encoded
Designing a Typed Scripting Language

- Design type system and execution model concurrently
- Be minimalistic: most features are encoded
- Use static near-equivalents for dynamic patterns
Designing a Typed Scripting Language

- Design type system and execution model concurrently
- Be minimalistic: most features are encoded
- Use static near-equivalents for dynamic patterns
- Infer all types: no type declarations
Designing a Typed Scripting Language

- Design type system and execution model concurrently
- Be minimalistic: most features are encoded
- Use static near-equivalents for dynamic patterns
- Infer all types: no type declarations
- Use a whole-program typechecking model
Designing a Typed Scripting Language

- Design type system and execution model concurrently
- Be minimalistic: most features are encoded
- Use static near-equivalents for dynamic patterns
- Infer all types: no type declarations
- Use a whole-program typechecking model
- Use type information to improve runtime memory layout
BigBang by Example
BigBang and TinyBang

• BigBang encodes to a core language
BigBang and TinyBang

- BigBang encodes to a core language
- TinyBang has very few features:
BigBang and TinyBang

- BigBang encodes to a core language
- TinyBang has very few features:
  - Primitives
BigBang and TinyBang

- BigBang encodes to a core language
- TinyBang has very few features:
  - Primitives
  - Labels
BigBang and TinyBang

- BigBang encodes to a core language
- TinyBang has very few features:
  - Primitives
  - Labels
  - Onions
BigBang and TinyBang

- BigBang encodes to a core language
- TinyBang has very few features:
  - Primitives
  - Labels
  - Onions
  - Scapes
BigBang and TinyBang

- BigBang encodes to a core language
- TinyBang has very few features:
  - Primitives
  - Labels
  - Onions
  - Scapes
  - Exceptions
BigBang and TinyBang

- BigBang encodes to a core language
- TinyBang has very few features:
  - Primitives
  - Labels
  - Onions
  - Scapes
  - Exceptions
  (and that’s all)
Labels and Onions

• Labels simply wrap data

```
'\texttt{name \ "Tom"}
```
Labels and Onions

- Labels simply wrap data (polymorphic variants)
- Onions combine data
- Data may be unlabeled (vs. extensible records)
- Onion data is projected by type
- Onioning is asymmetric (right-precedence)
  - Used to encode overriding
  - Important for type checking (later)

```plaintext
'name "Tom"
("Tom" ≠ 'name "Tom")
```
Labels and Onions

- Labels simply wrap data (polymorphic variants)
- Onions combine data

`'name "Tom" & 'age 10`
Labels and Onions

- Labels simply wrap data (polymorphic variants)
- Onions combine data
- Data may be unlabeled (vs. extensible records)

\['\text{name} \ "Tom" \ & \ \text{age} \ 10 \ & \ 3\]
Labels and Onions

- Labels simply wrap data (polymorphic variants)
- Onions combine data
- Data may be unlabeled (vs. extensible records)
- Onion data is projected by type

\[(1 \& (\_)) + 2 \implies 3\]
Labels and Onions

- Labels simply wrap data (polymorphic variants)
- Onions combine data
- Data may be unlabeled (vs. extensible records)
- Onion data is projected by type
- Onioning is asymmetric (right-precedence)

\[(1 \& 4) + 2 \implies 6\]
Labels and Onions

- Labels simply wrap data (polymorphic variants)
- Onions combine data
- Data may be unlabeled (vs. extensible records)
- Onion data is projected by type
- Onioning is asymmetric (right-precedence)
  - Used to encode overriding

\[(1 \& 4) + 2 \implies 6\]
Labels and Onions

- Labels simply wrap data (polymorphic variants)
- Onions combine data
- Data may be unlabeled (vs. extensible records)
- Onion data is projected by type
- Onioning is asymmetric (right-precedence)
  - Used to encode overriding
  - Important for type checking (later)

\[(1 \& 4) + 2 = 6\]
Scapes

- Scapes are functions

\[ x \rightarrow x \]
Scapes

- Scapes are functions with input patterns

`'A x & 'B y -> x + y`
Scapes

- Scapes are functions with input patterns

\[(\texttt{A} \ x \ & \ \texttt{B} \ y \ \rightarrow \ x \ + \ y) \ (\texttt{A} \ 1 \ & \ \texttt{B} \ 2) \Rightarrow 3\]
Scapes

- Scapes are functions with input patterns
- Onions of scapes apply the first matching scape

```plaintext
def list = 'Hd 4 &
    'Tl 'Nil () in
(('Hd h -> h) &
  ('Nil _ -> (())))
list
⇒ 4
```
Scapes

- Scapes are functions with input patterns
- Onions of scapes apply the first matching scape
- Encodes typecasing

\[
\text{def list} = \begin{cases}
\text{`Hd 4 & `Tl `Nil () in} \\
((\text{`Hd h -> h}) & \\
(\text{`Nil _ -> ()}))
\end{cases}
\]

\[
\Rightarrow
\text{case list of}
\begin{cases}
\text{`Nil _ -> ()} \\
\text{`Hd h -> h}
\end{cases}
\]
Scapes

- Scapes are functions with input patterns
- Onions of scapes apply the first matching scape
- Encodes typecasing
- Refines First-Class Cases [Chae et al. ’06]

```plaintext
def list = `Hd 4 & `Tl `Nil () in ((`Hd h -> h) & (`Nil _ -> ()
list
```
Mutation

- Label contents are mutable

```
def y = 'A 2 in
('A x -> x = 5 in y) y
⇒ 'A 5
```
Mutation

- Label contents are mutable
- But onioning is functional extension

```python
def x = 'A 0 & 'B 1 in
def y = 'B 2 & 'C 3 in
def z = x & y in
x

⇒ 'A 0 & 'B 1
```
Mutation

- Label contents are mutable
- But onioning is functional extension

```python
def x = 'A 0 & 'B 1 in
def y = 'B 2 & 'C 3 in
def z = x & y in
z

⇒ 'A 0 & 'B 2 & 'C 3
```
Expressiveness
Function self-awareness can be encoded by:

- Adding a `self` match to each pattern

\[
x \rightarrow x
\]

\[
\downarrow
\]

\[
x: 'self self \rightarrow x
\]
Encoding Self

Function self-awareness can be encoded by:

- Adding a ‘self match to each pattern

\[
\begin{align*}
\texttt{'A a -> e} \\
\downarrow \\
\texttt{'A a & 'self self -> e}
\end{align*}
\]
Encoding Self

Function self-awareness can be encoded by:

- Adding a 'self match to each pattern
- Adding a 'self value to each invocation

\[
\begin{align*}
\text{f } & \ e \\
\downarrow & \\
\text{f (e & 'self f)}
\end{align*}
\]
def factorial = x: int -> 
  if x == 0 then 1 else 
  self (x-1) * x
in self 5

⇓

def factorial = x: int & 'self self -> 
  if x == 0 then 1 else 
  self (x-1) * x
in factorial (5 & 'self factorial)
Objects are encoded as onions

```java
class Point {
    int x = 2;
    int y = 3;
    int l1() {
        return x+y;
    }
}
```
Encoding Objects

- Objects are encoded as onions
- Each field is a labeled value

```java
class Point {
    int x = 2;    // 'x 2
    int y = 3;    // 'y 3
    int l1() {
        return x+y;
    }
}
```
Encoding Objects

- Objects are encoded as onions
- Each field is a labeled value
- Message handler scapes encode methods

class Point {
    int x = 2;
    int y = 3;
    int l1() {
        return x+y;
    }
}

Encoding Objects

- Objects are encoded as onions
- Each field is a labeled value
- Message handler scapes encode methods

```java
class Point {
    int x = 2;
    int y = 3;
    int l1() {
        return x+y;
    }
}
```
Encoding Objects

def o =
    'x 2 &
    'y 3 &
    ( 'l1 () & 'self self ->
        self.x + self.y )
in

( 'x x -> x ) o ≃ o.x
def o =
    'x 2 &
    'y 3 &
    ( 'l1 () & 'self self ->
      self.x + self.y )
in

o ( 'l1 () & 'self o ) ≅ o.l1()
Encoding Mixins

- Inheritance occurs by onion extension

```python
def mypoint = 'x 2 & 'y 3 &
  ('l1 () -> self.x + self.y)
in def mixinFar =
  ('isFar () -> self.l1() > 26)
in def myFpoint = mypoint & mixinFar
in myFpoint.isFar()
```
Encoding Mixins

- Inheritance occurs by onion extension
- Mixins are the extension onion

```python
def mypoint = 'x 2 & 'y 3 &
   ('l1 () -> self.x + self.y)
in def mixinFar =
   ('isFar () -> self.l1() > 26)
in def myFpoint = mypoint & mixinFar
in myFpoint.isFar()```
Encoding Classes

• Classes are object factories

```python
def Point = 'new ('x x & 'y y) -> 'x x & 'y y &
    ('l1 () -> self.x + self.y)
in ...```
Encoding Classes

- Classes are object factories
- Subclass factories instantiate and extend

```python
def Point = 'new ('x x & 'y y) ->
    'x x & 'y y &
    ('l1 () -> self.x + self.y)

in def Point3D =
    'new (a: 'x _ & 'y _ & 'z z) ->
    def super = (Point.new a) in
    super & 'z 0 &
    ('l1 () -> super.l1()) + self.z)

in Point3D ('new ('x 1 & 'y 2 & 'z 3))
```
Encoding Overloading

- Overloading is trivial with scapes

```python
def join =
  (('x x:int & 'y y:int) -> x + y) &
  (('x _:unit & 'y _:unit) -> ())
in
join ('x 1 & 'y 2) & join ('x () & 'y ())
```
Encoding Overloading

- Overloading is trivial with scapes
- Onion extension allows incremental overloading

```python
def join = 
    ((`x x:int & `y y:int) -> x + y) &
    ((`x _:unit & `y _:unit) -> ())

in def x = join (`x 1 & `y 2) &
    join (`x () & `y ())

in def join = join &
    ((`x x:int & `y _:unit) -> x + 1)

in join (`x 5 & `y ())
```
Encoding Overloading

- Overloading is trivial with scapes
- Onion extension allows incremental overloading
- Default arguments are easy too

```python
def inc = a: `x x -> 
def by = ((_ -> 1) &
          (`y y -> y)) a
in x + by
in
inc (`x 1 & `y 2) + inc (`x 6)
```
Metaprogramming

- BigBang uses TinyBang as a core language
Metaprogramming

- BigBang uses TinyBang as a core language
- BigBang will provide macros for syntax/features
Metaprogramming

- BigBang uses TinyBang as a core language
- BigBang will provide macros for syntax/features
- `self`, class syntax, etc. defined in this way
Metaprogramming

- BigBang uses TinyBang as a core language
- BigBang will provide macros for syntax/features
- self, class syntax, etc. defined in this way
- User extensions can be specified
Metaprogramming

- BigBang uses TinyBang as a core language
- BigBang will provide macros for syntax/features
- `self`, class syntax, etc. defined in this way
- User extensions can be specified
- Similar to Racket (Languages as Libraries [Tobin-Hochstadt et al., 2011])
Typing
Typing Scripting Languages

A scripting language’s type system must be:

⭐ Expressive
Typing Scripting Languages

A scripting language’s type system must be:

★ Expressive
  • Duck typing, conditional types

Comprehensible
  • Types should be legible
  • Sources of type errors must be clear
  • Intuitive non-local inference

Efficient
  • Short compile times for dev. iterations

Easy to Use
  • Usable to teach introductory courses
Typing Scripting Languages

A scripting language’s type system must be:
⭐ Expressive
  - Duck typing, conditional types
  - No arbitrary cutoffs
Typing Scripting Languages

A scripting language’s type system must be:

⭐ Expressive
  - Duck typing, conditional types
  - No arbitrary cutoffs

✨ Comprehensible
Typing Scripting Languages

A scripting language’s type system must be:

🌟 Expressive
  - Duck typing, conditional types
  - No arbitrary cutoffs

✨ Comprehensible
  - Types should be legible
Typing Scripting Languages

A scripting language’s type system must be:

⭐ Expressive
  - Duck typing, conditional types
  - **No arbitrary cutoffs**

✨ Comprehensible
  - Types should be legible
  - **Sources of type errors must be clear**
Typing Scripting Languages

A scripting language’s type system must be:

🌟 Expressive
  - Duck typing, conditional types
  - **No arbitrary cutoffs**

✨ Comprehensible
  - Types should be legible
  - Sources of type errors must be clear
  - Intuitive non-local inference
Typing Scripting Languages

A scripting language’s type system must be:

⭐ Expressive
  - Duck typing, conditional types
  - **No arbitrary cutoffs**

✨ Comprehensible
  - Types should be legible
  - Sources of type errors must be clear
  - Intuitive non-local inference

🌟 Efficient
Typing Scripting Languages

A scripting language’s type system must be:

★ Expressive
  - Duck typing, conditional types
  - **No arbitrary cutoffs**

★ Comprehensible
  - Types should be legible
  - Sources of type errors must be clear
  - Intuitive non-local inference

★ Efficient
  - **Short compile times for dev. iterations**
Typing Scripting Languages

A scripting language’s type system must be:

⭐ Expressive
- Duck typing, conditional types
- **No arbitrary cutoffs**

✨ Comprehensible
- Types should be legible
- Sources of type errors must be clear
- Intuitive non-local inference

🌟 Efficient
- Short compile times for dev. iterations

✦ Easy to Use
Typing Scripting Languages

A scripting language’s type system must be:
★ Expressive
  • Duck typing, conditional types
  • No arbitrary cutoffs

★ Comprehensible
  • Types should be legible
  • Sources of type errors must be clear
  • Intuitive non-local inference

★ Efficient
  • Short compile times for dev. iterations

★ Easy to Use
  • Usable to teach introductory courses
Typing BigBang

For BigBang, we choose:

- ★ ✤ Subtype inference
- ★ Call-Site Polymorphism
- ★ ✦ Path sensitivity
- ✦ ✤ Flow insensitivity
- ★ ✉ Asymmetric concatenation
- ✉ Incremental typechecking

- ★ Expressive  ✤ Comprehensible
- ✉ Efficient  ✦ Easy to Use
Subtype Inference

- No programmer type declarations
Subtype Inference

- No programmer type declarations
- Supports duck-typing
Subtype Inference

- No programmer type declarations
- Supports duck-typing
- Supports nominal typing (labels as names)
Subtype Inference

- No programmer type declarations
- Supports duck-typing
- Supports nominal typing (labels as names)
  
  (e.g. `x 1 & `y 2 & `Point () )
Call-Site Polymorphism

- All functions polymorphic; no `let` restriction
Call-Site Polymorphism

- All functions polymorphic; no `let` restriction
- New contour for each non-recursive call site
Call-Site Polymorphism

- All functions polymorphic; no let restriction
- New contour for each non-recursive call site
- Only one contour for each recursive cycle
Call-Site Polymorphism

- All functions polymorphic; no `let` restriction
- New contour for each non-recursive call site
- Only one contour for each recursive cycle
- A variant of both \( n \)CFA and CPA
def f = x -> 'A x in
def x = f 0 in
def y = f () in
def z = f ('B 2 & 'C 3) in
...

Call-Site Polymorphism
Call-Site Polymorphism

```python
def f = x -> 'A x in
def x = f 0 in
def y = f () in
def z = f ('B 2 & 'C 3) in
...
```

\[
\begin{align*}
x & \implies 'A 0 \\
y & \implies 'A () \\
z & \implies 'A ('B 2 & 'C 3)
\end{align*}
\]
Path Sensitivity

- Scape application based on pattern match
Path Sensitivity

- Scape application based on pattern match
- Constraints expanded only if input matches
Path Sensitivity

- Scape application based on pattern match
- Constraints expanded only if input matches
- With polymorphism, gives path sensitivity
Path Sensitivity

- Scape application based on pattern match
- Constraints expanded only if input matches
- With polymorphism, gives path sensitivity
- Refines Conditional Types [Aiken et al. ’94]
Path Sensitivity

\[
def f = (\forall x \rightarrow x) \land (\forall y \rightarrow (\ )) \text{ in } f \ 'A 3
\]
Path Sensitivity

```python
def f = (\texttt{`A }x \rightarrow x) \land
    (\texttt{`B }y \rightarrow ()) \texttt{ in}
f \texttt{`A 3}

: \texttt{int}
```
Flow Insensitivity

- Type of a variable is flow-invariant
Flow Insensitivity

- Type of a variable is flow-invariant
- Flow sensitivity:
• Type of a variable is flow-invariant
• Flow sensitivity:
  • Makes variable types less clear
Flow Insensitivity

- Type of a variable is flow-invariant
- Flow sensitivity:
  - Makes variable types less clear
  - Brittle to refactoring
Flow Insensitivity

- Type of a variable is flow-invariant
- Flow sensitivity:
  - Makes variable types less clear
  - Brittle to refactoring
  - Doesn’t help that much
Flow Insensitivity

- Type of a variable is flow-invariant
- Flow sensitivity:
  - Makes variable types less clear
  - Brittle to refactoring
  - Doesn’t help that much

- Could be added later if needed
• In PL design, asymmetry can be good
• In PL design, **asymmetry can be good**
• Examples of asymmetry:
Asymmetric Concatenation

- In PL design, **asymmetry can be good**
- Examples of asymmetry:
  - Subtyping
Asymmetric Concatenation

- In PL design, **asymmetry can be good**
- Examples of asymmetry:
  - Subtyping
  - Overriding
Asymmetric Concatenation

- In PL design, **asymmetry can be good**
- Examples of asymmetry:
  - Subtyping
  - Overriding
  - Multiple inheritance
• In PL design, **asymmetry can be good**
• Examples of asymmetry:
  • Subtyping
  • Overriding
  • Multiple inheritance
  • Evaluation order
In PL design, **asymmetry can be good**

Examples of asymmetry:
- Subtyping
- Overriding
- Multiple inheritance
- Evaluation order
- Module dependencies
Asymmetric Concatenation

- Onion projection prefers rightmost element
Asymmetric Concatenation

- Onion projection prefers rightmost element
- **Not** based on row typing
Asymmetric Concatenation

- Onion projection prefers rightmost element
- **Not** based on row typing
- Only presence information is pushed forward
Asymmetric Concatenation

- Onion projection prefers rightmost element
- **Not** based on row typing
- Only presence information is pushed forward
  - Type system can express “α has ‘A int”

Upper bounds inferred from usage

Monomorphic variant of TinyBang closure is polynomial (vs. previous NP-complete result [Palsberg et al. '03])
Asymmetric Concatenation

- Onion projection prefers rightmost element
- **Not** based on row typing
- Only presence information is pushed forward
  - Type system can express "\(\alpha\) has \(\text{A int}\)"
  - But not "\(\alpha\) only has \(\text{A int}\)"

Upper bounds inferred from usage

Monomorphic variant of TinyBang closure is polynomial (vs. previous NP-complete result [Palsberg et al. '03])
Asymmetric Concatenation

- Onion projection prefers rightmost element
- **Not** based on row typing
- Only presence information is pushed forward
  - Type system can express “α has ‘A int”
  - But not “α only has ‘A int”
- Upper bounds inferred from usage
- Onion projection prefers rightmost element
- **Not** based on row typing
- Only presence information is pushed forward
  - Type system can express “α has ‘A int”
  - But not “α only has ‘A int”
- Upper bounds inferred from usage
- Monomorphic variant of TinyBang closure is polynomial (vs. previous NP-complete result [Palsberg et al. ’03])
Incremental Typechecking

- For scripts, edit-compile-debug must be fast
Incremental Typechecking

- For scripts, edit-compile-debug must be fast
- Type constraint closure can be slow

Solution:
- Track differences between software versions
- Delete constraints for removed code
- Include constraints from new code
- Perform closure again
Incremental Typechecking

- For scripts, edit-compile-debug must be fast
- Type constraint closure can be slow
- Solution:
Incremental Typechecking

- For scripts, edit-compile-debug must be fast
- Type constraint closure can be slow
- Solution:
  - Track differences between software versions
Incremental Typechecking

- For scripts, edit-compile-debug must be fast
- Type constraint closure can be slow
- Solution:
  - Track differences between software versions
  - Delete constraints for removed code
• For scripts, edit-compile-debug must be fast
• Type constraint closure can be slow
• Solution:
  • Track differences between software versions
  • Delete constraints for removed code
  • Include constraints from new code
Incremental Typechecking

- For scripts, edit-compile-debug must be fast
- Type constraint closure can be slow
- Solution:
  - Track differences between software versions
  - Delete constraints for removed code
  - Include constraints from new code
  - Perform closure again
Limitations

• Typical type system limitations
Limitations

- Typical type system limitations
  - Recursion limits contour creation
Limitations

- Typical type system limitations
  - Recursion limits contour creation
  - Flow-insensitivity
Limitations

- **Typical type system limitations**
  - Recursion limits contour creation
  - Flow-insensitivity
- **Syntactic limitations**
Limitations

• Typical type system limitations
  • Recursion limits contour creation
  • Flow-insensitivity

• Syntactic limitations
  • No string-to-label functionality
Compilation
Compilation

What will we want out of a compiler?

- Compiles scripts to native binaries (via LLVM)
- Optimizes layout using type information
- No unnecessary boxing
- Reduce pointer arithmetic
- Definitely no runtime hashing
- Still path-sensitive across modules
Compilation

What will we want out of a compiler?

- Compiles scripts to native binaries (via LLVM)
- Optimizes layout using type information
- No unnecessary boxing
- Reduce pointer arithmetic
- Definitely no runtime hashing
- Still path-sensitive across modules
Compilation

What will we want out of a compiler?

- Compiles scripts to native binaries (via LLVM)
- Optimizes layout using type information
  - No unnecessary boxing
- Reduce pointer arithmetic
- Definitely no runtime hashing
- Still path-sensitive across modules
Compilation

What will we want out of a compiler?

- Compiles scripts to native binaries (via LLVM)
- Optimizes layout using type information
  - No unnecessary boxing
  - Reduce pointer arithmetic
Compilation

What will we want out of a compiler?

- Compiles scripts to native binaries (via LLVM)
- Optimizes layout using type information
  - No unnecessary boxing
  - Reduce pointer arithmetic
- Definitely no runtime hashing
Compilation

What will we want out of a compiler?

• Compiles scripts to native binaries (via LLVM)
• Optimizes layout using type information
  • No unnecessary boxing
  • Reduce pointer arithmetic
  • Definitely no runtime hashing
• Still path-sensitive across modules
Compilation

What will we want out of a compiler?

• Compiles scripts to native binaries (via LLVM)
• Optimizes layout using type information
  • No unnecessary boxing
  • Reduce pointer arithmetic
  • Definitely no runtime hashing
• Still path-sensitive across modules

How do we get this?
Whole-Program Compilation!
Whole-Program Compilation

Why do we need a whole-program view?

- No declarations of types or module signatures
Why do we need a whole-program view?

- No declarations of types or module signatures
- General layout for extensible data structures is inefficient
Whole-Program Compilation

Why do we need a whole-program view?

- No declarations of types or module signatures
- General layout for extensible data structures is inefficient
- So we must know what could arrive at each call site
Whole-Program Compilation

How can we live with ourselves?

- Intermediate work (constraint sets, etc.) can be stored and reused
Whole-Program Compilation

How can we live with ourselves?

- Intermediate work (constraint sets, etc.) can be stored and reused
- Coding to a module signature is limited; not all interface semantics are typeable
Whole-Program Compilation

How can we live with ourselves?

- Intermediate work (constraint sets, etc.) can be stored and reused
- Coding to a module signature is limited; not all interface semantics are typeable
- Vast layout optimization potential
Whole-Program Compilation

How can we live with ourselves?

- Intermediate work (constraint sets, etc.) can be stored and reused
- Coding to a module signature is limited; not all interface semantics are typeable
- Vast layout optimization potential
- Shared libraries are still possible
Standard approach: common layout form (as C++)
• Standard approach: common layout form (as C++)
• Existing work handles flexible data structures
Layout

- Standard approach: common layout form (as C++)
- Existing work handles flexible data structures
  - A Calculus with Polymorphic and Polyvariant Flow Types [Wells et al. ’02]
Layout

- Standard approach: common layout form (as C++)
- Existing work handles flexible data structures
  - A Calculus with Polymorphic and Polyvariant Flow Types [Wells et al. ’02]
  - A Polymorphic Record Calculus and Its Compilation [Ohori ’95]
• Standard approach: common layout form (as C++)

• Existing work handles flexible data structures
  • A Calculus with Polymorphic and Polyvariant Flow Types [Wells et al. ’02]
  • A Polymorphic Record Calculus and Its Compilation [Ohori ’95]

• Onions: more flexible, new problems
• Standard approach: common layout form (as C++)
• Existing work handles flexible data structures
  • A Calculus with Polymorphic and Polyvariant Flow Types [Wells et al. ’02]
  • A Polymorphic Record Calculus and Its Compilation [Ohori ’95]
• Onions: more flexible, new problems
• Whole-program types will help us!
Where Are We?

We have:

- A TinyBang interpreter
Where Are We?

We have:

- A TinyBang interpreter
- A TinyBang type system and soundness proof

We need:

- A TinyBang-to-LLVM compiler
- A BigBang metaprogramming system
- A layout calculus and optimization tool
Where Are We?

We have:

- A TinyBang interpreter
- A TinyBang type system and soundness proof
- Effective encodings for language features

We need:

- A TinyBang-to-LLVM compiler
- A BigBang metaprogramming system
- A layout calculus and optimization tool
Where Are We?

We have:

- A TinyBang interpreter
- A TinyBang type system and soundness proof
- Effective encodings for language features

We need:
Where Are We?

We have:

• A TinyBang interpreter
• A TinyBang type system and soundness proof
• Effective encodings for language features

We need:

• A TinyBang-to-LLVM compiler
Where Are We?

We have:

- A TinyBang interpreter
- A TinyBang type system and soundness proof
- Effective encodings for language features

We need:

- A TinyBang-to-LLVM compiler
- A BigBang metaprogramming system
Where Are We?

We have:

- A TinyBang interpreter
- A TinyBang type system and soundness proof
- Effective encodings for language features

We need:

- A TinyBang-to-LLVM compiler
- A BigBang metaprogramming system
- A layout calculus and optimization tool
Where Are We?

We have:

- A TinyBang interpreter
- A TinyBang type system and soundness proof
- Effective encodings for language features

We need:

- A TinyBang-to-LLVM compiler
- A BigBang metaprogramming system
- A layout calculus and optimization tool

...
Questions?