Building a Typed Scripting Language

Zachary Palmer

The Johns Hopkins University

April 16th, 2015
Scripting: Good and Bad

```
1 print "What number?"
2 number = raw_input();
3 if number < 4:
4     print "Small!"
5 else:
6     print "Not small!"
```
Scripting: Good and Bad

```python
print "What number?"
number = raw_input();
if number < 4:
    print "Small!"
else:
    print "Not small!"
```

Scripting languages are...

- Terse and legible: easy to read and write
Scripting: Good and Bad

```python
print "What number?"
number = raw_input();
if number < 4:
    print "Small!"
else:
    print "Not small!"
```

Scripting languages are...

- Terse and legible: easy to read and write
- Flexible
Scripting: Good and Bad

Scripting languages are...

- Terse and legible: easy to read and write
- Flexible
- High-level

```python
print "What number?"
number = raw_input();
if number < 4:
    print "Small!"
else:
    print "Not small!"
```
Scripting: Good and Bad

```python
print "What number?"
number = raw_input();
if number < 4:
    print "Small!"
else:
    print "Not small!"
```

Scripting languages are...

- Terse and legible: easy to read and write
- Flexible
- High-level
- Error-prone
Scripting: Good and Bad

```java
1 import java.util.*;
2 public class SmallnessDetector {
3     public static void main(String[] arg) {
4         Scanner scanner = new Scanner(System.in);
5         System.out.println("What number?");
6         int number = scanner.nextLine();
7         if (number < 4) {
8             System.out.println("Small!");
9         } else {
10             System.out.println("Not small!");
11         }
12     }
13 }
```
import java.util.*;

public class SmallnessDetector {
    public static void main(String[] arg) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("What number?");
        int number = scanner.nextLine();
        if (number < 4) {
            System.out.println("Small!");
        } else {
            System.out.println("Not small!");
        }
    }
}

- Expression has type String
- Variable declared with type int
The Best of Both Worlds

Why can’t we just create a type system?
The Best of Both Worlds

Why can’t we just create a type system? We have.
The Best of Both Worlds

Why can’t we just create a type system? We have.

Rubydust

MyPy

TeJaS

Flow

Diamondback Ruby

PHP+QB

... ...

Hack

TypeScript
Retrofitting is Challenging

- Retrofitting: appealing, but very hard.
Retrofitting is Challenging

- Retrofitting: appealing, but very hard.
- Scripting languages: designed without type systems in mind.
Retrofitting is Challenging

- Retrofitting: appealing, but very hard.
- Scripting languages: designed without type systems in mind.

```python
1 x = 5
2 print x + 1
```
Retrofitting is Challenging

- Retrofitting: appealing, but very hard.
- Scripting languages: designed without type systems in mind.

```python
1   x = 5   int
2   print x + 1
```
Retrofitting is Challenging

- Retrofitting: appealing, but very hard.
- Scripting languages: designed without type systems in mind.

```python
1 x = 5  # int
2 print x + 1
```
Retrofitting is Challenging

- Retrofitting: appealing, but very hard.
- Scripting languages: designed without type systems in mind.

```python
1 x = 5
2 print x + 1
```

5/63
Retrofitting is Challenging

- Retrofitting: appealing, but very hard.
- Scripting languages: designed without type systems in mind.

```python
fs = [str, str.strip, len]
x = 5
locals()[raw_input()] = "foo"
print x + 1
exec(open(raw_input()).read())
```

5/63
Retrofitting is Challenging

- Retrofitting: appealing, but very hard.
- Scripting languages: designed without type systems in mind.

```python
fs = [str, str.strip, len]
xs = [True, " very ", "ab"]
for (f, x) in zip(fs, xs):
    print f(x)

x = 5
locals()[raw_input()] = "foo"
print x + 1
```
Retrofitting is Challenging

- Retrofitting: appealing, but very hard.
- Scripting languages: designed without type systems in mind.

```python
fs = [str, str.strip, len]
xs = [True, " very ", "ab"]
for (f,x) in zip(fs,xs):
    print f(x)
```

```python
x = 5
locals()[raw_input()] = "foo"
print x + 1
```

```python
exec(open(raw_input()).read())
```
Retrofitting is Challenging

- Retrofitting: appealing, but very hard.
- Scripting languages: designed without type systems in mind.

Observations:

- Fundamentally dynamic operations
Retrofitting is Challenging

- Retrofitting: appealing, but very hard.
- Scripting languages: designed without type systems in mind.

Observations:

- Fundamentally dynamic operations
- Contrived – not necessary for "scripting"
Retrofitting is Challenging

- Retrofitting: appealing, but very hard.
- Scripting languages: designed without type systems in mind.

Observations:

- Fundamentally dynamic operations
- Contrived – not necessary for “scripting”

Alternative: **build a typed scripting language from scratch**
Retrofitting is Challenging

- Retrofitting: appealing, but very hard.
- Scripting languages: designed without type systems in mind.

Observations:

- Fundamentally dynamic operations
- Contrived – not necessary for “scripting”

Alternative: **build a typed scripting language from scratch**
  - Include “scripty” expressiveness
Retrofitting is Challenging

- Retrofitting: appealing, but very hard.
- Scripting languages: designed without type systems in mind.

Observations:

- Fundamentally dynamic operations
- Contrived – not necessary for “scripting”

Alternative: **build a typed scripting language from scratch**

- Include “scripy” expressiveness
- Avoid dynamic operations
This Talk

Building a Typed Scripting Language
This Talk

Building a Typed Scripting Language

- Primary contribution: synthesis
This Talk

Building a Typed Scripting Language

- Primary contribution: synthesis
  - Underused type theory
This Talk

Building a Typed Scripting Language

- Primary contribution: synthesis
  - Underused type theory
  - Some abstract interpretation

Thesis:
It is possible to construct a language which has the static analyzability of traditional languages and the flexibility of scripting languages.
This Talk

Building a Typed Scripting Language

- Primary contribution: synthesis
  - Underused type theory
  - Some abstract interpretation
  - New type theory

Thesis:
It is possible to construct a language which has the static analyzability of traditional languages and the flexibility of scripting languages.
Building a Typed Scripting Language

- Primary contribution: synthesis
  - Underused type theory
  - Some abstract interpretation
  - New type theory

- Thesis: *It is possible to construct a language which has the static analyzability of traditional languages and the flexibility of scripting languages.*
Outline

- Duck Type Inference
- Conditional Reasoning
- Contextual Reasoning
- Flexible Data Model
- Formal Development
- What’s Left?
- Conclusion
Outline

- Duck Type Inference
- Conditional Reasoning
- Contextual Reasoning
- Flexible Data Model
- Formal Development
- What’s Left?
- Conclusion
Duck Typing

“When I see a bird that walks like a duck and swims like a duck
and quacks like a duck, I call that bird a duck.”

James Whitcomb Riley
Duck Typing

“When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.”

James Whitcomb Riley

Duck typing: categorizing data based on its exhibited properties (as opposed to by explicit grouping).
Java: No Duck Typing

```java
public interface Animal {
    public void speak();
}

public class Dog implements Animal {
    public void speak() {
        System.out.println("Woof!");
    }
}

Animal animal = new Dog();
animal.speak();
animal = new Sheep();
animal.speak();
```
Java: No Duck Typing

```java
public interface Animal {
    public void speak();
}

public class Dog implements Animal {
    public void speak() {
        System.out.println("Woof!");
    }
}

Animal animal = new Dog();
animal.speak();
animal = new Sheep();
animal.speak();
```
Python: Duck Typing

1 class Dog:
2     def speak(self): print "Woof!"
3 ...
4 animal = Dog();
5 animal.speak();
6 animal = Sheep();
7 animal.speak();
```python
class Dog:
    def speak(self): print "Woof!"
...
animal = Dog();
animal.speak();
animal = Sheep();
animal.speak();
```

![Diagram showing Duck Typing in Python]

Duck Typing in Python allows for a more flexible approach to object-oriented programming. In the example provided:

1. The `Dog` class has a `speak` method that prints "Woof!"
2. The `Sheep` class also has a `speak` method.
3. The `animal` variable can be assigned an instance of either `Dog` or `Sheep` and call the `speak` method.

This flexibility is a key aspect of Duck Typing, where the type of an object is determined by its methods rather than its class.
How do we type it?
Constraint Types

Let’s use constraint types!

Commonly used to type existing scripting languages.

Well-suited to type inference.

Type described by restrictions on variables.

\[ \alpha \{ \alpha \leq \text{has} \text{ speak()} \} \]

Expressive:

Union types:

\[ \alpha \{ \alpha \geq \text{int}, \alpha \geq \text{char} \} \]

Recursive types:

\[ \alpha_1 \{ \text{nil} \leq \alpha_1, \text{cons} \alpha_2 \alpha_1 \leq \alpha_1 \} \]

Inhabited iff deductive closure is consistent.

\[ \alpha_1 \{ \alpha_2 \geq \text{int}, \alpha_1 \geq \alpha_2, \text{char} \geq \alpha_1, \text{char} \geq \text{int} \} \]

\[ \alpha_1 \{ \alpha_2 \geq \text{int}, \alpha_1 \geq \alpha_2, \text{char} \geq \alpha_1, \text{char} \geq \text{int} \} \]

\[ \text{char} \geq \text{int} \text{ is false, so this type does not exist!} \]
Constraint Types

Let’s use constraint types!

- Commonly used to type existing scripting languages
Let’s use constraint types!

- Commonly used to type existing scripting languages
  - Well-suited to type inference
**Constraint Types**

Let’s use constraint types!

- Commonly used to type existing scripting languages
  - Well-suited to type inference
- Type described by restrictions on variables

\[
\alpha \{\alpha \leq \text{has speak()}}
\]

- Expressive:
  - Union types:
    \[
    \alpha \{\alpha \geq \text{int}, \alpha \geq \text{char}}
    \]
  - Recursive types:
    \[
    \alpha_1 \{\text{nil} \leq \alpha_1, \text{cons} \alpha_2 \alpha_1 \leq \alpha_1}
    \]

- Inhabited iff deductive closure is consistent

\[
\alpha_1 \{\alpha_2 \geq \text{int}, \alpha_1 \geq \alpha_2, \text{char} \geq \alpha_1, \text{char} \geq \text{int}}
\]

- char \geq \text{int} is false, so this type does not exist!
Constraint Types

Let’s use constraint types!

- Commonly used to type existing scripting languages
  - Well-suited to type inference

- Type described by restrictions on variables
  - \( \alpha \{ \alpha \leq \text{has speak()} \} \)

Expressive:
- Union types:
  - \( \alpha \{ \alpha \geq \text{int}, \alpha \geq \text{char} \} \)

- Recursive types:
  - \( \alpha_1 \{ \text{nil} \leq \alpha_1, \text{cons} \alpha_2 \alpha_1 \leq \alpha_1 \} \)

Inhabited iff deductive closure is consistent
- \( \alpha_1 \{ \alpha_2 \geq \text{int}, \alpha_1 \geq \alpha_2, \text{char} \geq \alpha_1, \text{char} \geq \text{int} \} \)

\( \text{char} \geq \text{int} \) is false, so this type does not exist!
Let’s use constraint types!

- Commonly used to type existing scripting languages
  - Well-suited to type inference
- Type described by restrictions on variables
  - $\alpha \{ \alpha \leq \text{has speak()} \}$

- Expressive:
  - Union types: $\alpha \{ \alpha \geq \text{int}, \alpha \geq \text{char} \}$
  - Recursive types: $\alpha_1 \{ \text{nil} \leq \alpha_1, \text{cons} \alpha_2 \alpha_1 \leq \alpha_1 \}$
  - etc.
Constraint Types

Let’s use constraint types!

- Commonly used to type existing scripting languages
  - Well-suited to type inference
- Type described by restrictions on variables
  - $\alpha \{ \alpha \leq \text{has speak()} \}$
- Expressive:
  - Union types: $\alpha \{ \alpha \geq \text{int}, \alpha \geq \text{char} \}$
  - Recursive types: $\alpha_1 \{ \text{nil} \leq \alpha_1, \text{cons } \alpha_2 \alpha_1 \leq \alpha_1 \}$
  - etc.
- Inhabited iff deductive closure is consistent
Constraint Types

Let’s use constraint types!

- Commonly used to type existing scripting languages
  - Well-suited to type inference
- Type described by restrictions on variables
  - $\alpha\{\alpha \leq \text{has speak()}\}$
- Expressive:
  - Union types: $\alpha\{\alpha \geq \text{int}, \alpha \geq \text{char}\}$
  - Recursive types: $\alpha_1\{\text{nil} \leq \alpha_1, \text{cons} \quad \alpha_2 \quad \alpha_1 \leq \alpha_1\}$
  - etc.

- Inhabited iff deductive closure is consistent
  - $\alpha_1\{\alpha_2 \geq \text{int}, \alpha_1 \geq \alpha_2, \text{char} \geq \alpha_1\}$
Constraint Types

Let’s use constraint types!

- Commonly used to type existing scripting languages
  - Well-suited to type inference
- Type described by restrictions on variables
  - $\alpha \{ \alpha \leq \text{has speak()} \}$

Expressive:
- Union types: $\alpha \{ \alpha \geq \text{int}, \alpha \geq \text{char} \}$
- Recursive types: $\alpha_1 \{ \text{nil} \leq \alpha_1, \text{cons } \alpha_2 \alpha_1 \leq \alpha_1 \}$
  - etc.

Inhabited iff deductive closure is consistent
- $\alpha_1 \{ \alpha_2 \geq \text{int}, \alpha_1 \geq \alpha_2, \text{char} \geq \alpha_1 \}$
- $\alpha_1 \{ \alpha_2 \geq \text{int}, \alpha_1 \geq \alpha_2, \text{char} \geq \alpha_1, \text{char} \geq \text{int} \}$
Let’s use constraint types!

- Commonly used to type existing scripting languages
  - Well-suited to type inference
- Type described by restrictions on variables
  - $\alpha\{\alpha \leq \text{has speak()}\}$
- Expressive:
  - Union types: $\alpha\{\alpha \geq \text{int}, \alpha \geq \text{char}\}$
  - Recursive types: $\alpha_1\{\text{nil} \leq \alpha_1, \text{cons } \alpha_2 \alpha_1 \leq \alpha_1\}$
  - etc.
- Inhabited iff deductive closure is consistent
  - $\alpha_1\{\alpha_2 \geq \text{int}, \alpha_1 \geq \alpha_2, \text{char} \geq \alpha_1\}$
  - $\alpha_1\{\alpha_2 \geq \text{int}, \alpha_1 \geq \alpha_2, \text{char} \geq \alpha_1, \text{char} \geq \text{int}\}$
  - char $\geq$ int is false, so this type does not exist!
Intuition

\[ \lambda x. (x > 4) \cdot (2 + 1) \]
Intuition

$1 \ (\lambda x. \ x > 4) \ (2+1)$
Intuition

\[ 1 \ (\lambda x. \ x>4) \ (2+1) \]
Intuition

$1 \ (\lambda x. \ x > 4) \ (2+1)$
Intuition

$1 \ (\lambda x. \ x > 4) \ (2+1)$
Intuition

1 (\( \lambda x. \ x > 4 \)) (2+1)

false
Intuition

\[ 1 \ (\lambda x. \ x > 4) \ (2 + 1) \]

false
Intuition

1 \((\lambda x. \; x>4) \; (2+1)\)

\[
\alpha_z \geq \alpha_f \alpha_a, \\
\alpha_a \geq \text{int} + \text{int} \\
\alpha_f \geq \alpha_x \rightarrow \alpha_r \{\alpha_r \geq \alpha_x > \text{int}\}
\]
Intuition

1 \((\lambda x. x > 4)(2+1)\)
Intuition

\[ 1 \ (\lambda x. \ x > 4) \ (2 + 1) \]
Intuition

$\lambda x. \ (x > 4) \ (2+1)$
Intuition

\[ 1 \ (\lambda x. \ x>4) \ (2+1) \]
Intuition

\((\lambda x. \ x > 4) (2+1)\)
Intuition

\[ (\lambda x. \ x > 4) \ (2+1) \]

Similar to abstract interpretation
Intuition

\[
\lambda x. \; (x > 4) \quad (2 + 1)
\]

- Similar to abstract interpretation
- Monotonic closure, invariant constraints
Outline

• Duck Type Inference
• Conditional Reasoning
• Contextual Reasoning
• Flexible Data Model
• Formal Development
• What’s Left?
• Conclusion
Conditional Reasoning

- Soundness of code based on case analysis
Conditional Reasoning

- Soundness of code based on case analysis
- Common tactic in scripting
Conditional Reasoning

- Soundness of code based on case analysis
- Common tactic in scripting
- Works particularly well with duck typing
def processHooks(tgt, data):
    if callable(tgt):
        fns = [tgt]
    else:
        fns = tgt
    for fn in fns:
        data = fn(data)
    return data
def processHooks(tgt, data):
    if callable(tgt):
        fns = [tgt]
    else:
        fns = tgt
    for fn in fns:
        data = fn(data)
    return data
def processHooks(tgt, data):
    if callable(tgt):
        fns = [tgt]
    else:
        fns = tgt
    for fn in fns:
        data = fn(data)
    return data
def processHooks(tgt, data):
    if callable(tgt):
        fns = [tgt]
    else:
        fns = tgt
    for fn in fns:
        data = fn(data)
    return data
def processHooks(tgt, data):
    if callable(tgt):
        fns = [tgt]
    else:
        fns = tgt
    for fn in fns:
        data = fn(data)
    return data
def processHooks(tgt, data):
    if callable(tgt):
        fns = [tgt]
    else:
        fns = tgt
    for fn in fns:
        data = fn(data)
    return data
def processHooks(tgt, data):
    if callable(tgt):
        fns = [tgt]
    else:
        fns = tgt
    for fn in fns:
        data = fn(data)
    return data
def processHooks(tgt, data):
    if callable(tgt):
        fns = [tgt]
    else:
        fns = tgt
    for fn in fns:
        data = fn(data)
    return data

function list
function

Program analyses with conditional reasoning: "path-sensitive"
def processHooks(tgt, data):
    if callable(tgt):
        fns = [tgt]
    else:
        fns = tgt
    for fn in fns:
        data = fn(data)
    return data
def processHooks(tgt, data):
    if callable(tgt):
        fns = [tgt]
    else:
        fns = tgt
    for fn in fns:
        data = fn(data)
    return data
def processHooks(tgt, data):
    if callable(tgt):
        fns = [tgt]
    else:
        fns = tgt
    for fn in fns:
        data = fn(data)
    return data

Program analyses with conditional reasoning: "path-sensitive"
How do we type it?
Typing Path-Sensitivity

- Soundness reasoning by case analysis on value
Typing Path-Sensitivity

- Soundness reasoning by case analysis on value
- Thus, use case analysis types
Typing Path-Sensitivity

- Soundness reasoning by case analysis on value
- Thus, use case analysis types
- Use only those constraints associated with correct case
Typing Path-Sensitivity

- Soundness reasoning by case analysis on value
- Thus, use case analysis types
- Use only those constraints associated with correct case

```haskell
1 let x = 4 in
2 case x of
3     int -> 0
4     char -> 'a'
```
Typing Path-Sensitivity

- Soundness reasoning by case analysis on value
- Thus, use case analysis types
- Use only those constraints associated with correct case

```haskell
let x = 4 in
case x of
  int -> 0
  char -> 'a'
```
Typing Path-Sensitivity

- Soundness reasoning by case analysis on value
- Thus, use case analysis types
- Use only those constraints associated with correct case

```haskell
1 let x = 4 in
2 case x of
3   int -> 0
4   char -> 'a'
```
Typing Path-Sensitivity

- Soundness reasoning by case analysis on value
- Thus, use case analysis types
- Use only those constraints associated with correct case

```
let x = 4 in
case x of
  int -> 0
  char -> 'a'
```
Typing Path-Sensitivity

- Soundness reasoning by case analysis on value
- Thus, use case analysis types
- Use only those constraints associated with correct case

```plaintext
1 let x = 4 in
2 case x of
3     int -> 0
4     char -> 'a'
```
Outline

- Duck Type Inference
- Conditional Reasoning
  - Filtered Types
- Contextual Reasoning
- Flexible Data Model
- Formal Development
- What’s Left?
- Conclusion
Filtered Types

1 let x = (if somebool then 4 else 'z') in
2 case x of
3    int -> x + 1
4    z    -> 0
let x = (if somebool then 4 else 'z') in

case x of
  int -> x + 1
  z -> 0

known to be an int!
Filtered Types

```haskell
let x = (if somebool then 4 else 'z') in

case x of
  int -> x + 1  
  z -> 0

# We want refinement on case analysis
```

known to be an int!
Filtered Types

```ocaml
let x = (if somebool then 4 else 'z') in

case x of

  int  -> x + 1
  z    -> 0

known to be an int!
```

- We want refinement on case analysis, but
  - We'd like to preserve the invariance of types
Filtered Types

```
let x = (if somehowbool then 4 else 'z') in

case x of
  int -> x + 1
  z -> 0
```

- We want refinement on case analysis, but
  - We’d like to preserve the invariance of types and
  - We have to keep decidability in mind
Filtered Types

1 let x = (if somebool then 4 else 'z') in
2 case x of
3     y * int -> y + 1
4     z -> 0
Filtered Types

```
let x = (if somebool then 4 else 'z') in

case x of
  y * int -> y + 1
  z -> 0

α_y ≥ α_x ∩ int
```
Filtered Types

1 let x = (if somebool then 4 else 'z') in
2 case x of
3    y * int -> y + 1
4    z -> 0

- $\alpha_y \geq \alpha_x \cap \text{int}$
- General intersections are infeasible [?]

Also includes negation (top level only): $\neg$
Filtered Types

1. let x = (if somebool then 4 else 'z') in
2. case x of
3.     y int --> y + 1
4.     z --> 0

- $\alpha_y \geq \alpha_x \cap \text{int}$
- General intersections are infeasible [?]  
- Filtered types: $\tau \cap \pi \cap \pi \cap \ldots$
let x = (if somebool then 4 else 'z') in

case x of
  y * int -> y + 1
  z -> 0

- $\alpha_y \geq \alpha_x \cap \text{int}$
- General intersections are infeasible [?]
- Filtered types: $\tau \cap \pi \cap \pi \cap \ldots$
- Also includes negation (top level only): $\alpha_z \geq 1 - \text{int}$
def processHooks(tgt, data):
    let fns =
    case tgt of
        f * fun -> [f]
        x -> x
    for fn in fns:
        fn(data)
    return data
def processHooks(tgt, data):
    let fns =
        case tgt of
            f * fun -> [f]
            x -> x
        for fn in fns:
            fn(data)
    return data
def processHooks(tgt, data):
    let fns =
    case tgt of
        f * fun -> [f]
        x -> x
    for fn in fns:
        fn(data)
    return data
Filtered Types

```python
def processHooks(tgt, data):
    let fns =
        case tgt of
            f * fun -> [f]
            x -> x
    for fn in fns:
        fn(data)
    return data
```

fun ∪ [fun]

(fun ∪ [fun]) ∩ fun = fun
Filtered Types

```python
def processHooks(tgt, data):
    let fns =
    case tgt of
        f * fun -> [f]
        x -> x
    for fn in fns:
        fn(data)
    return data
```

1. `fun ∪ [fun]`
2. `(fun ∪ [fun]) ∩ fun = fun`
3. `(fun ∪ [fun]) ∩ (1 − fun)`
def processHooks(tgt, data):
    let fns =
    case tgt of
        f * fun -> [f]
        x -> x
    for fn in fns:
        fn(data)
    return data
Outline

- Duck Type Inference
- Conditional Reasoning
  - Filtered Types
- Contextual Reasoning
- Flexible Data Model
- Formal Development
- What’s Left?
- Conclusion
Contextual Reasoning

1 def strange(b):
2     if b: return (lambda n: n+1)
3     else: return 4
4 f = strange(True)
5 x = strange(False)
6 print f(x)

- f is a function from integers to integers
Contextual Reasoning

```python
def strange(b):
    if b: return (lambda n: n+1)
    else: return 4

f = strange(True)
x = strange(False)
print f(x)
```

- $f$ is a function from integers to integers
- $x$ is a single integer value
Contextual Reasoning

```python
def strange(b):
    if b: return (lambda n: n+1)
    else: return 4
f = strange(True)
x = strange(False)
print f(x)
```

- \( f \) is a function from integers to integers
- \( x \) is a single integer value
- Return types are different based on invocation context
Contextual Reasoning

```python
def strange(b):
    if b: return (lambda n: n+1)
    else: return 4
f = strange(True)
x = strange(False)
print f(x)
```

- $f$ is a function from integers to integers
- $x$ is a single integer value
- Return types are different based on invocation context
- Program analyses with contextual reasoning: “context-sensitive”
How do we type it?
Context Sensitivity

- Let-bound polymorphism
- Existing program analyses [?]
- Conditional constraints [?]
Context Sensitivity

- Let-bound polymorphism (too weak)
- Existing program analyses [\(?\)]
- Conditional constraints [\(?\)]
Context Sensitivity

- Let-bound polymorphism (too weak)
- Existing program analyses [?] (too brittle, not suited)
- Conditional constraints [?]
Context Sensitivity

- Let-bound polymorphism (too weak)
- Existing program analyses [?] (too brittle, not suited)
- Conditional constraints [?] (too coarse)
Context Sensitivity

- Let-bound polymorphism (too weak)
- Existing program analyses [?] (too brittle, not suited)
- Conditional constraints [?] (too coarse)
- Call-site polymorphism [?, ?]
Call-Site Polymorphism

- **All** functions inferred polymorphic types
Call-Site Polymorphism

- All functions inferred polymorphic types
- Polymethod instantiation at call sites
Call-Site Polymorphism

- **All** functions inferred polymorphic types
- Polyinstantiation at call sites
- **Expressiveness:**

  ```
  1 let f x = 1  def f() = fun y -> y
  2   let g y = y in 2
  3    g;; 3
  4 let h = f();; 4  def h = f()
  5 let q = (h 1, h 'z');; 5  def q = (h 1, h 'z')
  ```
Bounding Call-Site Polymorphism

\[ 1 \ (\lambda x. \ x \ x) \ (\lambda x. \ x \ x) \]
Bounding Call-Site Polymorphism

$1 \ (\lambda x. \ x \ x) \ (\lambda x. \ x \ x)$
Bounding Call-Site Polymorphism

\[ 1 \ (\lambda x. \ x \ x) \ (\lambda x. \ x \ x) \]
Bounding Call-Site Polymorphism

$$1 \ (\lambda x. \ x \ x) \ (\lambda x. \ x \ x)$$
Bounding Call-Site Polymorphism

Variables named by program point and contour

Contour: restricted regular expression

Concatenation of literals and Kleene closures over literals

Each literal token may appear at most once

Contours are joined at recursion

\[
x \cdot \cdot \cdot a \cdot \cdot \cdot b
\]

\[
x : b
\]

\[
\epsilon
\]

\[
ax : ba \cdot \cdot \cdot a : ba
\]
Bounding Call-Site Polymorphism

Variables named by program point and contour

Contour: restricted regular expression
Concatenation of literals and Kleene closures over literals
Each literal token may appear at most once

Contours are joined at recursion

\[ x: b \]
\[ a: b \]
\[ b: \epsilon \]
Bounding Call-Site Polymorphism

Variables named by program point and contour

Contour: restricted regular expression

Concatenation of literals and Kleene closures over literals

Each literal token may appear at most once

\[
x \cdot x:b \\
\cdot \cdot \\
a:b
\]

\[
x:b \\
\cdot \cdot \\
a:b \\
\]

\[
x:b \\
\cdot \cdot \\
a:b \\
\]

...
Bounding Call-Site Polymorphism

Variables named by program point and contour

Contour: restricted regular expression

Concatenation of literals and Kleene closures over literals

Each literal token may appear at most once

Contours are joined at recursion

\[ \epsilon \]
Variables named by program point and contour
Bounding Call-Site Polymorphism

- Variables named by program point and **contour**
- Contour: restricted regular expression

![Diagram](image-url)
Bounding Call-Site Polymorphism

- Variables named by program point and **contour**
- Contour: restricted regular expression
  - Concatenation of literals and Kleene closures over literals
Bounding Call-Site Polymorphism

- Variables named by program point and **contour**
- Contour: restricted regular expression
  - Concatenation of literals and Kleene closures over literals
  - Each literal token may appear at most once
Variables named by program point and **contour**

- Contour: restricted regular expression
  - Concatenation of literals and Kleene closures over literals
  - Each literal token may appear at most once

- Contours are joined at recursion
Outline

- Duck Type Inference
- Conditional Reasoning
  - Filtered Types
- Contextual Reasoning
- Flexible Data Model
- Formal Development
- What’s Left?
- Conclusion
Flexible Data Model

Scripting languages have flexible data models:

```python
class MyClass:
    def msg(self):
        print "Foo"

obj = MyClass()
obj.msg() # Prints "Foo"
obj.msg = types.MethodType(lambda s: print "Bar", obj)
obj.msg() # Prints "Bar"
```
Flexible Data Model

Scripting languages have flexible data models:

```python
class MyClass:
    def msg(self):
        print "Foo"

obj = MyClass()
obj.msg()  # Prints "Foo"

obj.msg = types.MethodType(lambda s: print "Bar", obj)

obj.msg()  # Prints "Bar"
```

- Mutating/adding methods
- Multiple inheritance
- Dynamic mixins
- etc.
How do we type it?
Encoding

- Reduces complexity of type system
Encoding

- Reduces complexity of type system
- Ensures consistency in reasoning
Encoding

- Reduces complexity of type system
- Ensures consistency in reasoning
- No artificial distinctions (e.g. strategy object \( \cong \) function)
Encoding

- Reduces complexity of type system
- Ensures consistency in reasoning
- No artificial distinctions (e.g. strategy object $\simeq$ function)
- Programmer can work “under the hood” as necessary
Encoding

LittleBang

tinyBang

Encode

TinyBang

Typecheck

Typechecking Result

35/63
TinyBang Language Features

- Primitives (e.g. 5)
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
- Partial functions
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
- Partial functions
  - fun int -> 0 matches any integer
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
- Partial functions
  - `fun int -> 0` matches any integer
  - `fun x:int -> x` is integer identity

(fun x:int -> x + 1) 4 returns 5
(fun x -> x + 1) "badness" crashes
(fun int -> 0) "badness" also crashes
(fun 'A x -> x) removes an ‘A label
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
- Partial functions
  - `fun int -> 0` matches any integer
  - `fun x:int -> x` is integer identity
  - `(fun x:int -> x + 1) 4` returns 5
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
- Partial functions
  - `fun int -> 0` matches any integer
  - `fun x:int -> x` is integer identity
  - `(fun x:int -> x + 1) 4` returns 5
  - `(fun x -> x + 1) "badness"` crashes
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. 'A 5)
- Partial functions
  - `fun int -> 0` matches any integer
  - `fun x:int -> x` is integer identity
  - `(fun x:int -> x + 1) 4` returns 5
  - `(fun x -> x + 1) "badness"` crashes
  - `(fun int -> 0) "badness"` also crashes
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
- Partial functions
- Onions: type-indexed records [?]
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
- Partial functions
- Onions: type-indexed records [?]
  - “Normal” records: {Name="Ann", Age=43}
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
- Partial functions
- Onions: type-indexed records
  - “Normal” records: {Name="Ann", Age=43}
  - 4 & "word" is an onion with an int and a str
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
- Partial functions
- Onions: type-indexed records [?]
  - “Normal” records: {Name="Ann", Age=43}
  - 4 & "word" is an onion with an int and a str
  - Onions of labels: records/structs
    - ‘Name "Ann" & ‘Age 43
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
- Partial functions
- Onions: type-indexed records
  - “Normal” records: {Name="Ann", Age=43}
  - 4 & "word" is an onion with an int and a str
  - Onions of labels: records/structs
    - ‘Name "Ann" & ‘Age 43
  - Functions match onions by type
    - (fun ‘Name n -> n) (‘Name "Ann" & ‘Age 43) returns "Ann"
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
- Partial functions
- Onions: type-indexed records [?]
  - “Normal” records: {Name="Ann", Age=43}
  - 4 & "word" is an onion with an int and a str
  - Onions of labels: records/structs
    - ‘Name "Ann" & ‘Age 43
  - Functions match onions by type
    - (fun ‘Name n -> n) (‘Name "Ann" & ‘Age 43) returns "Ann"
    - **Leftmost** onion element wins
      - (fun ‘A x -> x) (‘A 2 & ‘B 3 & ‘A 4) returns 2
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
- Partial functions
- Onions: type-indexed records [？]
- Onion dispatch: leftmost function wins
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
- Partial functions
- Onions: type-indexed records [?]
- Onion dispatch: leftmost function wins
  - (int -> 0) & (char -> ’a’) matches int or char onions
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
- Partial functions
- Onions: type-indexed records 
- Onion dispatch: leftmost function wins
  - (int -> 0) & (char -> ’a’) matches int or char onions
  - ((int -> 0) & (char -> ’a’)) (5) \(\Rightarrow\) 0
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
- Partial functions
- Onions: type-indexed records [?]
- Onion dispatch: leftmost function wins
  
  (int -> 0) & (char -> ’a’) matches int or char onions
  ((int -> 0) & (char -> ’a’)) (5) ⇒ 0
  ((int -> 0) & (char -> ’a’)) (’z’) ⇒ ’a’
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
- Partial functions
- Onions: type-indexed records [?]
- Onion dispatch: leftmost function wins
  - \((\text{int} \rightarrow 0) \& (\text{char} \rightarrow 'a')\) matches \text{int} or \text{char} onions
  - \(((\text{int} \rightarrow 0) \& (\text{char} \rightarrow 'a'))\) (5) \(\Rightarrow\) 0
  - \(((\text{int} \rightarrow 0) \& (\text{char} \rightarrow 'a'))\) ('z') \(\Rightarrow\) 'a'
  - \(((\text{int} \rightarrow 0) \& (\text{char} \rightarrow 'a'))\) (5 \& 'z') \(\Rightarrow\) 0
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
- Partial functions
- Onions: type-indexed records [?]
- Onion dispatch: leftmost function wins
  - \((\text{int} \rightarrow 0) \& (\text{char} \rightarrow 'a')\) matches \text{int} or \text{char} onions
  - \(((\text{int} \rightarrow 0) \& (\text{char} \rightarrow 'a'))\) (5) \(\rightarrow\) 0
  - \(((\text{int} \rightarrow 0) \& (\text{char} \rightarrow 'a'))\) ('z') \(\rightarrow\) 'a'
  - \(((\text{int} \rightarrow 0) \& (\text{char} \rightarrow 'a'))\) (5 & 'z') \(\rightarrow\) 0
  - \(((\text{int} \rightarrow 0) \& (\text{char} \rightarrow 'a'))\) (‘A 5 & ‘B 4) crashes
TinyBang Language Features

- Primitives (e.g. 5)
- Labels (e.g. ‘A 5)
- Partial functions
- Onions: type-indexed records
- Onion dispatch: leftmost function wins
- That’s it!
Why These Features?

Together, onions and partial functions can encode:
Why These Features?

Together, onions and partial functions can encode:

- Records
- Conditionals (on `True ()` and `False ()`)
- Variant-based objects
- Operator overloading
- Classes, inheritance, subclasses, etc.
- Mixins, dynamic functional object extension
- First-class cases
- Optional arguments
- etc.
Why These Features?

Together, onions and partial functions can encode:

- Records
- Conditionals (on `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!
Encoding Example

LittleBang

1  def inc(x,y=1):
2      return x + y
3  print inc(3,5)
4  print inc(7)
Encoding Example

**LittleBang**
1. `def inc(x,y=1):
2.     return x + y
3. print inc(3,5)
4. print inc(7)

**TinyBang**
1. `let inc = fun a * 'x x ->
2.     let y = ( (fun 'y v -> v)
3.         & (fun _ -> 1)   ) a
4.     in x + y
5. print (inc ('x 3 & 'y 5))
6. print (inc ('x 7))`
Working Under the Hood

```python
let obj = if somebool
    then object {
      m(s:str) = print(s)
    }
    else object {
      inc(x:int) = x + 1
    }
obj.m("hello") # static type error if no m

def dynamic(msg): throw MethodError()
(obj & dynamic).m("hello") # exception if no m
```
let obj = if somebool
    then
        fun ('msg 'm () * 's (s * str)) -> print(s)
    else
        fun ('msg 'inc () * 'x (x * int)) -> x + 1

obj ('msg 'm () & 's "hello") # static type error if no m

let dynamic = fun msg -> throw MethodError()
(obj & dynamic) ('msg 'm () & 's "hello") # exception if no m
What We Don’t Get

TinyBang is:
- Aware of conditionals
- Aware of calling context
- Flexible on data
What We Don’t Get

TinyBang is:
- Aware of conditionals
- Aware of calling context
- Flexible on data

TinyBang isn’t:
- Perfect on recursion
- Modular
- Aware of time
What We Don’t Get

TinyBang is:
- Aware of conditionals
- Aware of calling context
- Flexible on data

TinyBang isn’t:
- Perfect on recursion
- Modular
- Aware of time

```python
def c = 4
c = "hello"
(str -> "") c
```
What We Don’t Get

TinyBang is:

- Aware of conditionals
- Aware of calling context
- Flexible on data

TinyBang isn’t:

- Perfect on recursion
- Modular
- Aware of time

- No typestate

```python
1 def c = 4
2 c = "hello"
3 (str -> "") c
```
What We Don’t Get

TinyBang is:
- Aware of conditionals
- Aware of calling context
- Flexible on data

TinyBang isn’t:
- Perfect on recursion
- Modular
- Aware of time

1  def c = 4
2  c = "hello"
3  (str -> ")") c

- No typestate
- No mutable monkeypatching (use functional extension!)
Outline

- Duck Type Inference
- Conditional Reasoning
  - Filtered Types
- Contextual Reasoning
- Flexible Data Model
- Formal Development
- What’s Left?
- Conclusion
Typechecking Process

To typecheck a LittleBang program:

- Encode LittleBang operations into TinyBang
- A-translate TinyBang into A-normal form
- Like assembly language: lots of small steps
- Derive type constraints from program
  e.g. $x = 5$ implies $\alpha x \geq \text{int}$
- Perform deductive logical closure
  Reach every conclusion we can from what we've learned
- Similar to running the program
- Check consistency of result
  If the program is bad, we will reach a false conclusion
Typechecking Process

To typecheck a LittleBang program:

- Encode LittleBang operations into TinyBang
To typecheck a LittleBang program:

- Encode LittleBang operations into TinyBang
- $A$-translate TinyBang into $A$-normal form
Typechecking Process

To typecheck a LittleBang program:

- Encode LittleBang operations into TinyBang
- \textit{A-translate} TinyBang into A-normal form
  - Like assembly language: lots of small steps

Like
\[ x = 5 \implies \alpha x \geq \text{int} \]

Derive type constraints from program.

Perform deductive logical closure.

Reach every conclusion we can from what we've learned.

Similar to running the program.

Check consistency of result.

If the program is bad, we will reach a false conclusion.
To typecheck a LittleBang program:

- Encode LittleBang operations into TinyBang
- $A$-translate TinyBang into A-normal form
  - Like assembly language: lots of small steps
Typechecking Process

To typecheck a LittleBang program:

- Encode LittleBang operations into TinyBang
- \textit{A-translate} TinyBang into A-normal form
  - Like assembly language: lots of small steps

\begin{tikzpicture}[->,auto,font=\footnotesize]
  
  \node[rectangle,fill=green!30,draw] (l) at (0,0) {LittleBang};
  \node[rectangle,fill=yellow!30,draw] (t) at (2,0) {TinyBang ANF};
  \node[rectangle,fill=orange!30,draw] (r) at (4,0) {Typechecking Result};
  \node[rectangle,fill=blue!30,draw] (e) at (0,-1) {encoded};
  \node[rectangle,fill=blue!30,draw] (a) at (2,-1) {A-translate};
  \node[rectangle,fill=blue!30,draw] (t) at (4,-1) {typecheck};

  \draw (l) edge node {} (e)
        (e) edge node {} (a)
        (a) edge node {} (t)
        (t) edge node {} (r)
        (r) edge node {} (t);
\end{tikzpicture}
Typechecking Process

To typecheck a LittleBang program:

- Encode LittleBang operations into TinyBang
- A-\textit{translate} TinyBang into A-normal form
  - Like assembly language: lots of small steps
- Derive type constraints from program
Typechecking Process

To typecheck a LittleBang program:

- Encode LittleBang operations into TinyBang
- $A$-translate TinyBang into A-normal form
  - Like assembly language: lots of small steps
- Derive type constraints from program
  - e.g. $x=5$ implies $\alpha_x \geq \text{int}$
Typechecking Process

To typecheck a LittleBang program:

- Encode LittleBang operations into TinyBang
- \textit{A-translate} TinyBang into A-normal form
  - Like assembly language: lots of small steps
- Derive type constraints from program
  - e.g. $x=5$ implies $\alpha_X \geq \text{int}$
- Perform deductive logical closure
Typechecking Process

To typecheck a LittleBang program:

- Encode LittleBang operations into TinyBang
- \textit{A-translate} TinyBang into A-normal form
  - Like assembly language: lots of small steps
- Derive type constraints from program
  - e.g. \( x=5 \) implies \( \alpha_x \geq \text{int} \)
- Perform deductive logical closure
  - Reach every conclusion we can from what we’ve learned
Typechecking Process

To typecheck a LittleBang program:

- Encode LittleBang operations into TinyBang
- **A-translate** TinyBang into A-normal form
  - Like assembly language: lots of small steps
- Derive type constraints from program
  - e.g. $x=5$ implies $\alpha_x \geq \text{int}$
- Perform deductive logical closure
  - Reach every conclusion we can from what we’ve learned
  - Similar to running the program
Typechecking Process

To typecheck a LittleBang program:

- Encode LittleBang operations into TinyBang
- \textit{A-translate} TinyBang into A-normal form
  - Like assembly language: lots of small steps
- Derive type constraints from program
  - e.g. \( x=5 \) implies \( \alpha_x \geq \text{int} \)
- Perform deductive logical closure
  - Reach every conclusion we can from what we’ve learned
  - Similar to running the program
- Check consistency of result
Typechecking Process

To typecheck a LittleBang program:

- Encode LittleBang operations into TinyBang
- *A-translation* TinyBang into A-normal form
  - Like assembly language: lots of small steps
- Derive type constraints from program
  - e.g. \( x=5 \) implies \( \alpha_x \geq \text{int} \)
- Perform deductive logical closure
  - Reach every conclusion we can from what we’ve learned
  - Similar to running the program
- Check consistency of result
  - If the program is bad, we will reach a false conclusion
TinyBang type grammar is shallow (unusual)

- e.g. $\alpha_1 \{ \text{`A } \alpha_2 \leq \alpha_1, \text{int} \leq \alpha_2 \}$
- not `A int
Synthesis

- TinyBang type grammar is shallow (unusual)
  - e.g. $\alpha_1 \{ 'A \alpha_2 \leq \alpha_1, \text{int} \leq \alpha_2 \}$
  - not $'A \text{ int}$

- Corresponds to A-normalized expression grammar
TinyBang type grammar is shallow (unusual)

- e.g. $\alpha_1 \{ ‘A \alpha_2 \leq \alpha_1, \text{int} \leq \alpha_2 \}$
- not ‘A int

Corresponds to A-normalized expression grammar

In general, correspondence between evaluation and type systems

- value $\leftrightarrow$ type
- clause $\leftrightarrow$ constraint
- expression $\leftrightarrow$ constraint type
- operational semantics rule $\leftrightarrow$ constraint closure rule
- evaluation $\leftrightarrow$ constraint closure

Forms adjoint (as in abstract interpretation)

Proof of soundness: simulation rather than progress & preservation

Difference: all facts gathered here are invariant
TinyBang type grammar is shallow (unusual)

- e.g. \( \alpha_1 \backslash \{ 'A \alpha_2 \leq \alpha_1, \text{int} \leq \alpha_2 \} \)
- not 'A int

Corresponds to A-normalized expression grammar

In general, correspondence between evaluation and type systems

- value ↔ type
- clause ↔ constraint
- expression ↔ constraint type
- operational semantics rule ↔ constraint closure rule
- evaluation ↔ constraint closure

Forms adjoint (as in abstract interpretation)
Synthesis

- TinyBang type grammar is shallow (unusual)
  - e.g.  $\alpha_1 \{ 'A \alpha_2 \leq \alpha_1, \text{int} \leq \alpha_2 \}$
  - not $'A \text{int}$

- Corresponds to A-normalized expression grammar

In general, correspondence between evaluation and type systems

- value $\leftrightarrow$ type
- clause $\leftrightarrow$ constraint
- expression $\leftrightarrow$ constraint type
- operational semantics rule $\leftrightarrow$ constraint closure rule
- evaluation $\leftrightarrow$ constraint closure

- Forms adjoint (as in abstract interpretation)

- Proof of soundness: simulation rather than progress & preservation
Synthesis

- TinyBang type grammar is shallow (unusual)
  - e.g. $\alpha_1 \{ 'A \alpha_2 \leq \alpha_1, \text{int} \leq \alpha_2 \}$
  - not '$A \text{int}$

- Corresponds to A-normalized expression grammar

- In general, correspondence between evaluation and type systems
  - value $\leftrightarrow$ type
  - clause $\leftrightarrow$ constraint
  - expression $\leftrightarrow$ constraint type
  - operational semantics rule $\leftrightarrow$ constraint closure rule
  - evaluation $\leftrightarrow$ constraint closure

- Forms adjoint (as in abstract interpretation)

- Proof of soundness: simulation rather than progress & preservation

- Difference: all facts gathered here are invariant
Proof of Soundness

Progress & Preservation
Proof of Soundness

Progress & Preservation

\[ \vdash: e_0 \rightarrow \tau_0 \]

\[ \vdash: \geq \]

\[ \vdash: \leq \]
Proof of Soundness

Progress & Preservation

\[ \tau_0 \vdash: e_0 \rightarrow e_1 \]
Proof of Soundness

Progress & Preservation

\[ \vdash: e_0 \rightarrow e_1 \]

\[ \vdash: \tau_0 \rightarrow \tau_1 \]
Proof of Soundness

Progress & Preservation

\[ \vdash : e_0 \rightarrow e_1 \]

\[ \vdash : \tau_0 \geq \tau_1 \]
Proof of Soundness

Progress & Preservation

\[ \vdash: e_0 \rightarrow e_1 \]

\[ \vdash: \tau_0 \geq \rightarrow \tau_1 \]

\[ \vdash: \cdots \]
Proof of Soundness

Simulation

\[ \vdash : \]

\[ \tau_0 \rightarrow \tau_1 \]

\[ e_0 \rightarrow e_1 \]

\[ \vdash : \]
Proof of Soundness

Simulation

\[ \tau_0 \quad \vartriangleright \quad \tau_1 \]

\[ e_0 \quad \vartriangleright \quad e_1 \]

\[ \vdash : \]

\[ \vdash : \]

\[ \Rightarrow \]
Proof of Soundness

Simulation

\[ \vdash e_0 \rightarrow e_1 \]

\[ \vdash \tau_0 \rightarrow \tau_1 \]

\[ \vdash \ldots \]
Union alignment invariants
Not Shown Here

- Union alignment invariants
- The interesting encodings (dynamic mixins, subclasses, etc.)
Union alignment invariants

The interesting encodings (dynamic mixins, subclasses, etc.)

Novel object extension properties [?]
Outline

- Duck Type Inference
- Conditional Reasoning
  - Filtered Types
- Contextual Reasoning
- Flexible Data Model
- Formal Development
- What’s Left?
- Conclusion
What’s Left?

- Current Research

- Function patterns (e.g. \(\text{int} \rightarrow \text{int}\))
  - Encode type signatures
  - First overloading of higher-order functions
  - Potentially solves modularity issues

- Chronological constraints
  - Enable flow-sensitivity (awareness of time)
  - Possibly improves typechecking performance

- Layout calculus
  - Hari’s dissertation focus
  - Efficient onion memory layout and dispatch

- Future Research
  - Type error processing
  - Modularity
  - User-specified encodings / language features
  - Type-aware script metaprogramming
What’s Left?

Current Research

- Function patterns (e.g. `int ~> int`)
What’s Left?

Current Research

- Function patterns (e.g. `int ~> int`)
  - Encode type signatures
What’s Left?

Current Research

- Function patterns (e.g. int ~> int)
  - Encode type signatures
  - First overloading of higher-order functions

Future Research

- Type error processing
- Modularity
- User-specified encodings / language features
- Type-aware script metaprogramming
What’s Left?

Current Research
- Function patterns (e.g. `int ~> int`)
  - Encode type signatures
  - First overloading of higher-order functions
  - Potentially solves modularity issues

Future Research
- Type error processing
- Modularity
- User-specified encodings / language features
- Type-aware script metaprogramming

Layout calculus
- Hari’s dissertation focus
  - Efficient onion memory layout and dispatch

Chronological constraints
- Enable flow-sensitivity (awareness of time)
  - Possibly improves typechecking performance
What’s Left?

- **Current Research**
  - Function patterns (e.g. `int ~> int`)
    - Encode type signatures
    - First overloading of higher-order functions
    - Potentially solves modularity issues
  - **Chronological constraints**

- **Future Research**
  - Type error processing
  - Modularity
  - User-specified encodings / language features
  - Type-aware script metaprogramming

- **Hari's dissertation focus**
  - Efficient onion memory layout and dispatch

- **Layout calculus**
  - Enable flow-sensitivity (awareness of time)
  - Possibly improves typechecking performance
What’s Left?

- **Current Research**
  - Function patterns (e.g. `int ~> int`)
    - Encode type signatures
    - First overloading of higher-order functions
    - Potentially solves modularity issues
  - Chronological constraints
    - Enable flow-sensitivity (awareness of time)
What’s Left?

- **Current Research**
  - Function patterns (e.g. \texttt{int} \mapsto \texttt{int})
    - Encode type signatures
    - First overloading of higher-order functions
    - Potentially solves modularity issues
  - Chronological constraints
    - Enable flow-sensitivity (awareness of time)
    - Possibly improves typechecking performance

- **Future Research**

What’s Left?

Current Research
- Function patterns (e.g. \texttt{int} \rightarrow \texttt{int})
  - Encode type signatures
  - First overloading of higher-order functions
  - Potentially solves modularity issues
- Chronological constraints
  - Enable flow-sensitivity (awareness of time)
  - Possibly improves typechecking performance
- Layout calculus
What’s Left?

- Current Research
  - Function patterns (e.g. `int ~> int`)
    - Encode type signatures
    - First overloading of higher-order functions
    - Potentially solves modularity issues
  - Chronological constraints
    - Enable flow-sensitivity (awareness of time)
    - Possibly improves typechecking performance
  - Layout calculus
    - Hari’s dissertation focus

- Future Research
  - Type error processing
  - Modularity
  - User-specified encodings / language features
  - Type-aware script metaprogramming
What’s Left?

Current Research

- Function patterns (e.g. \texttt{int} \rightarrow \texttt{int})
  - Encode type signatures
  - First overloading of higher-order functions
  - Potentially solves modularity issues

- Chronological constraints
  - Enable flow-sensitivity (awareness of time)
  - Possibly improves typechecking performance

- Layout calculus
  - Hari’s dissertation focus
  - Efficient onion memory layout and dispatch
What’s Left?

- **Current Research**
  - Function patterns (e.g. `int ~> int`)
    - Encode type signatures
    - First overloading of higher-order functions
    - Potentially solves modularity issues
  - Chronological constraints
    - Enable flow-sensitivity (awareness of time)
    - Possibly improves typechecking performance
  - Layout calculus
    - Hari’s dissertation focus
    - Efficient onion memory layout and dispatch

- **Future Research**
What's Left?

Current Research
- Function patterns (e.g. `int ~> int`)
  - Encode type signatures
  - First overloading of higher-order functions
  - Potentially solves modularity issues
- Chronological constraints
  - Enable flow-sensitivity (awareness of time)
  - Possibly improves typechecking performance
- Layout calculus
  - Hari's dissertation focus
  - Efficient onion memory layout and dispatch

Future Research
- Type error processing
What’s Left?

Current Research
- Function patterns (e.g. `int ~> int`)
  - Encode type signatures
  - First overloading of higher-order functions
  - Potentially solves modularity issues
- Chronological constraints
  - Enable flow-sensitivity (awareness of time)
  - Possibly improves typechecking performance
- Layout calculus
  - Hari’s dissertation focus
  - Efficient onion memory layout and dispatch

Future Research
- Type error processing
- Modularity
What’s Left?

- **Current Research**
  - Function patterns (e.g. `int ~> int`)
    - Encode type signatures
    - First overloading of higher-order functions
    - Potentially solves modularity issues
  - Chronological constraints
    - Enable flow-sensitivity (awareness of time)
    - Possibly improves typechecking performance
  - Layout calculus
    - Hari’s dissertation focus
    - Efficient onion memory layout and dispatch

- **Future Research**
  - Type error processing
  - Modularity
  - **User-specified encodings / language features**
What’s Left?

- **Current Research**
  - Function patterns (e.g. `int ~> int`)
    - Encode type signatures
    - First overloading of higher-order functions
    - Potentially solves modularity issues
  - Chronological constraints
    - Enable flow-sensitivity (awareness of time)
    - Possibly improves typechecking performance
  - Layout calculus
    - Hari’s dissertation focus
    - Efficient onion memory layout and dispatch

- **Future Research**
  - Type error processing
  - Modularity
  - User-specified encodings / language features
  - **Type-aware script metaprogramming**
Outline

- Duck Type Inference
- Conditional Reasoning
  - Filtered Types
- Contextual Reasoning
- Flexible Data Model
- Formal Development
- What’s Left?
- Conclusion
Conclusions

We can build a typed scripting language from scratch. Subtype constraints allow for duck typing. Feature encodings keep type system simple. Asymmetry of onions allows encoding of subclasses, overloading, etc. Path-sensitivity (via onion dispatch) enables case-based reasoning. Context-sensitivity (via call-site polymorphism) allows complex, intuitive soundness arguments. Exploit connection to abstract interpretation while remaining in type theory.
Conclusions

We can build a typed scripting language from scratch.
Conclusions

We can build a typed scripting language from scratch.

- Subtype constraints allow for duck typing
Conclusions

We can build a typed scripting language from scratch.

- Subtype constraints allow for duck typing
- Feature encodings keep type system simple
Conclusions

We can build a typed scripting language from scratch.

- Subtype constraints allow for duck typing
- Feature encodings keep type system simple
  - Asymmetry of onions allows encoding of subclasses, overloading, etc.
Conclusions

We can build a typed scripting language from scratch.

- Subtype constraints allow for duck typing
- Feature encodings keep type system simple
  - Asymmetry of onions allows encoding of subclasses, overloading, etc.
- Path-sensitivity (via onion dispatch) enables case-based reasoning
Conclusions

We can build a typed scripting language from scratch.

- Subtype constraints allow for duck typing
- Feature encodings keep type system simple
  - Asymmetry of onions allows encoding of subclasses, overloading, etc.
- Path-sensitivity (via onion dispatch) enables case-based reasoning
- Context-sensitivity (via call-site polymorphism) allows complex, intuitive soundness arguments
Conclusions

We can build a typed scripting language from scratch.

- Subtype constraints allow for duck typing
- Feature encodings keep type system simple
  - Asymmetry of onions allows encoding of subclasses, overloading, etc.
- Path-sensitivity (via onion dispatch) enables case-based reasoning
- Context-sensitivity (via call-site polymorphism) allows complex, intuitive soundness arguments
- Exploit connection to abstract interpretation while remaining in type theory
Thanks!

- Scott F. Smith (advisor)
- Alexander Rozenstheyn (collaborator)
- Pottayil Harisanker Menon (collaborator)
- Rebekah Palmer (wife, best friend)
- JHU Computer Science Department
- All those people who did all that research
Questions?
Typechecking Example
Typechecking by Example

First, we will typecheck this program:

1  let b = ...
2  1 + 2 + (if b then 5 else 1)
Typechecking by Example

First, we will typecheck this program:

```ml
let b = ...
1 + 2 + (if b then 5 else 1)
```

Then, we will typecheck this program:

```ml
let b = ...
1 + 2 + (if b then 'z' else 1)
```
Encoding and A-normalization
Preparing to Typecheck

LittleBang

1 let b = ... 
2 1 + 2 + (if b then 5 else 1)
Preparing to Typecheck

LittleBang

1 let b = ...
2 1 + 2 + (if b then 5 else 1)

TinyBang

1 let b = ... in
2 1 + 2 + ( ((‘True () -> 5) & (‘False () -> 1)) b)
Preparing to Typecheck

LittleBang

1 let b = ...
2 1 + 2 + (if b then 5 else 1)

TinyBang

1 let b = ... in
2 1 + 2 + ( ((‘True () -> 5)
3       & (‘False () -> 1)) b)

TinyBang

1 b = ...;
2 x1 = 1;
3 x2 = 2;
4 x3 = + x1 x2;
5 x4 = { p1 = (); p2 = ‘True p1 } -> { r1 = 5 };
6 x5 = { p3 = (); p4 = ‘False p3 } -> { r2 = 1 };
7 x6 = x4 & x5;
8 x7 = x6 b;
9 x8 = + x3 x7;

ANF
An Aside: Execution

1  \( b = \ldots; \)
2  \( x_1 = 1; \)
3  \( x_2 = 2; \)
4  \( x_3 = + x_1 \times x_2; \)
5  \( x_4 = \{ p_1 = (); p_2 = 'True p_1 \} \rightarrow \{ r_1 = 5 \}; \)
6  \( x_5 = \{ p_3 = (); p_4 = 'False p_3 \} \rightarrow \{ r_2 = 1 \}; \)
7  \( x_6 = x_4 \& x_5; \)
8  \( x_7 = x_6 \ b; \)
9  \( x_8 = + x_3 \times x_7; \)
An Aside: Execution

1. `b = ...;
2. x1 = 1;
3. x2 = 2;
4. x3 = 3;
5. x4 = { p1 = (); p2 = 'True p1' } -> { r1 = 5 };
6. x5 = { p3 = (); p4 = 'False p3' } -> { r2 = 1 };
7. x6 = x4 & x5;
8. x7 = x6 b;
9. x8 = + x3 x7;
An Aside: Execution

1. \( b = 'False ...; \)
2. \( x1 = 1; \)
3. \( x2 = 2; \)
4. \( x3 = 3; \)
5. \( x4 = \{ p1 = (); p2 = 'True p1 } \rightarrow \{ r1 = 5 \}; \)
6. \( x5 = \{ p3 = (); p4 = 'False p3 } \rightarrow \{ r2 = 1 \}; \)
7. \( x6 = x4 \& x5; \)
8. \( x7 = x6 b; \)
9. \( x8 = + x3 x7; \)
An Aside: Execution

1 \( b = \text{`False` ...}; \)
2 \( x_1 = 1; \)
3 \( x_2 = 2; \)
4 \( x_3 = 3; \)
5 \( x_4 = \{ p_1 = (); p_2 = \text{`True p1` } \rightarrow \{ r_1 = 5 \}; \)
6 \( x_5 = \{ p_3 = (); p_4 = `False p3` } \rightarrow \{ r_2 = 1 \};\)
7 \( x_6 = x_4 \& x_5; \)
8 \( x_7 = x_6 b; \)
9 \( x_8 = + x_3 x_7; \)
An Aside: Execution

1  \texttt{b = 'False ...;}
2  \texttt{x1 = 1;}
3  \texttt{x2 = 2;}
4  \texttt{x3 = 3;}
5  \texttt{x4 = \{ p1 = (); p2 = 'True p1 \} -> \{ r1 = 5 \};}
6  \texttt{x5 = \{ p3 = (); p4 = 'False p3 \} -> \{ r2 = 1 \};}
7  \texttt{x6 = x4 \& x5;}
8  \texttt{x7 = x6 b;}
9  \texttt{x8 = + x3 x7;}

An Aside: Execution

1  b = 'False ...;
2  x1 = 1;
3  x2 = 2;
4  x3 = 3;
5  x4 = { p1 = (); p2 = 'True p1 } -> { r1 = 5 };
6  x5 = { p3 = (); p4 = 'False p3 } -> { r2 = 1 };
7  x6 = x4 & x5;
8  x7 = x6 b;
9  x8 = + x3 x7;
An Aside: Execution

1. \( b = \text{‘False ...} \);
2. \( x1 = 1; \)
3. \( x2 = 2; \)
4. \( x3 = 3; \)
5. \( x4 = \{ p1 = (); \ p2 = \text{‘True p1 } \} \rightarrow \{ r1 = 5 \}; \)
6. \( x5 = \{ p3 = (); \ p4 = \text{‘False p3 } \} \rightarrow \{ r2 = 1 \}; \)
7. \( x6 = x4 \& x5; \)
8. \( x7 = x6 \ b; \)
9. \( x8 = + x3 \ x7; \)
An Aside: Execution

1. \( b = \text{'False'} \ldots; \)
2. \( x1 = 1; \)
3. \( x2 = 2; \)
4. \( x3 = 3; \)
5. \( x4 = \{ p1 = (); p2 = \text{'True \ p1'} \} \rightarrow \{ r1 = 5 \}; \)
6. \( x5 = \{ p3 = (); p4 = \text{'False \ p3'} \} \rightarrow \{ r2 = 1 \}; \)
7. \( x6 = x4 \& x5; \)
8. \( r2' = 1; x7 = r2'; \)
9. \( x8 = + x3 \ x7; \)
An Aside: Execution

1. $b = \text{'False ...;}$
2. $x1 = 1;$
3. $x2 = 2;$
4. $x3 = 3;$
5. $x4 = \{ p1 = (); \ p2 = \text{'True p1 } \} \rightarrow \{ r1 = 5 \};$
6. $x5 = \{ p3 = (); \ p4 = \text{'False p3 } \} \rightarrow \{ r2 = 1 \};$
7. $x6 = x4 \& x5;$
8. $r2' = 1; \ x7 = r2'$
9. $x8 = + x3 \ x7;$
An Aside: Execution

1. $b = 'False ...$;
2. $x1 = 1$;
3. $x2 = 2$;
4. $x3 = 3$;
5. $x4 = \{ p1 = (); p2 = 'True p1 } \rightarrow \{ r1 = 5 \}$;
6. $x5 = \{ p3 = (); p4 = 'False p3 } \rightarrow \{ r2 = 1 \}$;
7. $x6 = x4 \& x5$;
8. $r2' = 1$; $x7 = 1$;
9. $x8 = + x3 x7$;
An Aside: Execution

1. $b = \text{False \ldots}$
2. $x_1 = 1$
3. $x_2 = 2$
4. $x_3 = 3$
5. $x_4 = \{ p_1 = (); p_2 = \text{True p1} \} \rightarrow \{ r_1 = 5 \}$
6. $x_5 = \{ p_3 = (); p_4 = \text{False p3} \} \rightarrow \{ r_2 = 1 \}$
7. $x_6 = x_4 \& x_5$
8. $r_2' = 1; x_7 = 1$
9. $x_8 = 4$
Initial Alignment

1  b = ...;
2  x1 = 1;
3  x2 = 2;
4  x3 = + x1 x2;
5  x4 = { p1 = (); p2 = 'True p1 } -> { r1 = 5 };
6  x5 = { p3 = (); p4 = 'False p3 } -> { r2 = 1 };
7  x6 = x4 & x5;
8  x7 = x6 b;
9  x8 = + x3 x7;
Initial Alignment

```
1  b = ...;
2  x1 = int;
3  x2 = int;
4  x3 = + x1 x2;
5  x4 = { p1 = (); p2 = 'True p1 } -> { r1 = int };
6  x5 = { p3 = (); p4 = 'False p3 } -> { r2 = int };
7  x6 = x4 & x5;
8  x7 = x6 b;
9  x8 = + x3 x7;
```

- Replace primitive data with its type
Initial Alignment

\[
\begin{align*}
\alpha_b \geq & \ 'True' \ldots, \quad \alpha_b \geq \ 'False' \ldots, \\
\alpha_{x1} \geq & \ \text{int}, \\
\alpha_{x2} \geq & \ \text{int}, \\
\alpha_{x3} \geq & \ + \ \alpha_{x1} \ \alpha_{x2}, \\
\alpha_{x4} \geq & \ \{\alpha_{p1} \geq () , \alpha_{p2} \geq \ 'True' \ \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \ \text{int}\}, \\
\alpha_{x5} \geq & \ \{\alpha_{p3} \geq () , \alpha_{p4} \geq \ 'False' \ \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \ \text{int}\}, \\
\alpha_{x6} \geq & \ \alpha_{x4} \ \& \ \alpha_{x5}, \\
\alpha_{x7} \geq & \ \alpha_{x6} \ \alpha_{b}, \\
\alpha_{x8} \geq & \ + \ \alpha_{x3} \ \alpha_{x7},
\end{align*}
\]

- Replace primitive data with its type
- Convert to constraints (for e.g. duck typing)
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq 'True \ldots, \quad \alpha_b \geq 'False \ldots, \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq ()\, , \alpha_{p2} \geq 'True \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{int}\}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq ()\, , \alpha_{p4} \geq 'False \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_b, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b &\geq 'True ..., \quad \alpha_b \geq 'False ...,
\alpha_{x1} &\geq \text{int}, \\
\alpha_{x2} &\geq \text{int}, \\
\alpha_{x3} &\geq + \alpha_{x1} \cdot \alpha_{x2}; \\
\alpha_{x4} &\geq \{\alpha_{p1} \geq () , \alpha_{p2} \geq 'True \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{int}\}, \\
\alpha_{x5} &\geq \{\alpha_{p3} \geq () , \alpha_{p4} \geq 'False \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} &\geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} &\geq \alpha_{x6} \cdot \alpha_b, \\
\alpha_{x8} &\geq + \alpha_{x3} \cdot \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq 'True ...,
\alpha_b & \geq 'False ...,
\alpha_{x1} & \geq \text{int},
\alpha_{x2} & \geq \text{int},
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2},
\alpha_{x4} & \geq \{\alpha_{p1} \geq () , \alpha_{p2} \geq 'True \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{int}\},
\alpha_{x5} & \geq \{\alpha_{p3} \geq () , \alpha_{p4} \geq 'False \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\},
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5},
\alpha_{x7} & \geq \alpha_{x6} \alpha_b,
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq 'True \ldots, \quad \alpha_b \geq 'False \ldots, \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}; \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq () \}, \alpha_{p2} \geq 'True \alpha_{p1} \rightarrow \{\alpha_{r1} \geq \text{int}\}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq () \}, \alpha_{p4} \geq 'False \alpha_{p3} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_b, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b \geq \text{‘True ...}, & \quad \alpha_b \geq \text{‘False ...}, \\
\alpha_{x1} \geq \text{int}, & \\
\alpha_{x2} \geq \text{int}, & \\
\alpha_{x3} \geq + \alpha_{x1} \alpha_{x2}, & \\
\alpha_{x4} \geq \{\alpha_{p1} \geq () , \alpha_{p2} \geq \text{‘True } \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{int}\}, & \\
\alpha_{x5} \geq \{\alpha_{p3} \geq () , \alpha_{p4} \geq \text{‘False } \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, & \\
\alpha_{x6} \geq \alpha_{x4} \& \alpha_{x5}, & \\
\alpha_{x7} \geq \alpha_{x6} \alpha_b, & \\
\alpha_{x8} \geq + \alpha_{x3} \alpha_{x7} & \\
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq 'True \ldots, \quad \alpha_b \geq 'False \ldots, \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq (), \alpha_{p2} \geq 'True \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{int}\}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq (), \alpha_{p4} \geq 'False \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_b, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[ \begin{align*}
\alpha_b &\geq 'True \ldots, \quad \alpha_b \geq 'False \ldots, \\
\alpha_{x1} &\geq \text{int}, \\
\alpha_{x2} &\geq \text{int}, \\
\alpha_{x3} &\geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} &\geq \{\alpha_{p1} \geq ()\}, \alpha_{p2} \geq 'True \alpha_{p1} \rightarrow \{\alpha_{r1} \geq \text{int}\}, \\
\alpha_{x5} &\geq \{\alpha_{p3} \geq ()\}, \alpha_{p4} \geq 'False \alpha_{p3} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} &\geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} &\geq \alpha_{x6} \alpha_{b}, \\
\alpha_{x8} &\geq + \alpha_{x3} \alpha_{x7}
\end{align*} \]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq \text{`True ...}, \quad \alpha_b \geq \text{`False ...}, \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq () , \alpha_{p2} \geq \text{`True } \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{int}\}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq () , \alpha_{p4} \geq \text{`False } \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_b, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq \text{`True ...}, \quad \alpha_b \geq \text{`False ...}, \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}; \\
\alpha_{x4} & \geq \{ \alpha_{p1} \geq () , \alpha_{p2} \geq \text{`True } \alpha_{p1} \} \rightarrow \{ \alpha_{r1} \geq \text{int} \}, \\
\alpha_{x5} & \geq \{ \alpha_{p3} \geq () , \alpha_{p4} \geq \text{`False } \alpha_{p3} \} \rightarrow \{ \alpha_{r2} \geq \text{int} \}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_b, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{cases}
\alpha_b \geq 'True...', \quad \alpha_b \geq 'False...,' \\
\alpha_{x1} \geq \text{int}, \\
\alpha_{x2} \geq \text{int}, \\
\alpha_{x3} \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} \geq \{\alpha_{p1} \geq () \}, \alpha_{p2} \geq 'True \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{int}\}, \\
\alpha_{x5} \geq \{\alpha_{p3} \geq () \}, \alpha_{p4} \geq 'False \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} \geq \alpha_{x6} \alpha_{b}, \\
\alpha_{x8} \geq + \alpha_{x3} \alpha_{x7}
\end{cases}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq 'True ...', \quad \alpha_b \geq 'False ...', \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{ \alpha_{p1} \geq () , \alpha_{p2} \geq 'True \alpha_{p1} \} \rightarrow \{ \alpha_{r1} \geq \text{int} \}, \\
\alpha_{x5} & \geq \{ \alpha_{p3} \geq () , \alpha_{p4} \geq 'False \alpha_{p3} \} \rightarrow \{ \alpha_{r2} \geq \text{int} \}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_{b}, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq 'True ...,' \quad \alpha_b \geq 'False ...,' \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq () , \alpha_{p2} \geq 'True \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{int}\}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq () , \alpha_{p4} \geq 'False \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_b, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq 'True \ldots', \quad \alpha_b \geq 'False \ldots', \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq ()\}, \quad \alpha_{p2} \geq 'True \alpha_{p1}\} \to \{\alpha_{r1} \geq \text{int}\}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq ()\}, \quad \alpha_{p4} \geq 'False \alpha_{p3}\} \to \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_b, \quad \alpha_{x7} \geq \text{int}, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b &\geq \text{'True } \ldots, \quad \alpha_b &\geq \text{'False } \ldots, \\
\alpha_{x1} &\geq \text{int}, \\
\alpha_{x2} &\geq \text{int}, \\
\alpha_{x3} &\geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} &\geq \{\alpha_{p1} \geq () , \alpha_{p2} \geq \text{'True } \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{int}\}, \\
\alpha_{x5} &\geq \{\alpha_{p3} \geq () , \alpha_{p4} \geq \text{'False } \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} &\geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} &\geq \alpha_{x6} \alpha_b, \quad \alpha_{x7} &\geq \text{int}, \\
\alpha_{x8} &\geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{aligned}
\alpha_b & \geq 'True \ldots', \quad \alpha_b \geq 'False \ldots', \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq ()\}, \alpha_{p2} \geq 'True \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{int}\}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq ()\}, \alpha_{p4} \geq 'False \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_b, \quad \alpha_{x7} \geq \text{int}, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{aligned}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq 'True \ldots, \quad \alpha_b \geq 'False \ldots, \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq () , \alpha_{p2} \geq 'True \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{int}\}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq () , \alpha_{p4} \geq 'False \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} & \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_b, \quad \alpha_{x7} \geq \text{int}, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}, \\
\end{align*}
\]
Check Consistency

\[
\begin{align*}
\alpha_b &\geq \text{`True ...}, \quad \alpha_b \geq \text{`False ...}, \\
\alpha_{x1} &\geq \text{int}, \\
\alpha_{x2} &\geq \text{int}, \\
\alpha_{x3} &\geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} &\geq \{\alpha_{p1} \geq () , \alpha_{p2} \geq \text{`True } \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{int}\}, \\
\alpha_{x5} &\geq \{\alpha_{p3} \geq () , \alpha_{p4} \geq \text{`False } \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} &\geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} &\geq \alpha_{x6} \alpha_b, \quad \alpha_{x7} \geq \text{int}, \\
\alpha_{x8} &\geq + \alpha_{x3} \alpha_{x7},
\end{align*}
\]
Typechecking by Example

First, we will typecheck this program:

1  let b = ...
2  1 + 2 + (if b then 5 else 1)

Then, we will typecheck this program:

1  let b = ...
2  1 + 2 + (if b then 'z' else 1)
b = ...;

x1 = 1;

x2 = 2;

x3 = + x1 x2;

x4 = { p1 = (); p2 = 'True p1 } -> { r1 = 'z' };

x5 = { p3 = (); p4 = 'False p3 } -> { r2 = 1 };

x6 = x4 & x5;

x7 = x6 b;

x8 = + x3 x7;
Initial Alignment

\[
\begin{align*}
\alpha_b & \geq 'True \ldots, \quad \alpha_b \geq 'False \ldots, \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq ()\}, \alpha_{p2} \geq 'True \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{char}\}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq ()\}, \alpha_{p4} \geq 'False \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_{b}, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7},
\end{align*}
\]

• Same as last time, but with a char
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq 'True \ldots, \quad \alpha_b \geq 'False \ldots, \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq +\alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq ()}, \alpha_{p2} \geq 'True \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{char}\}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq ()}, \alpha_{p4} \geq 'False \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_{b}, \\
\alpha_{x8} & \geq +\alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq \text{"True..."} , \quad \alpha_b \geq \text{"False..."}, \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}; \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq () , \alpha_{p2} \geq \text{"True } \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{char}\}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq () , \alpha_{p4} \geq \text{"False } \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_{b}, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b \geq \text{'True ...}, \\
\alpha_x \geq \text{int}, \\
\alpha_x \geq \text{int}, \\
\alpha_x \geq + \alpha_x \alpha_x; \\
\alpha_x \geq \{\alpha_p \geq () , \alpha_p \geq \text{'True } \alpha_p \} \rightarrow \{\alpha_r \geq \text{char}\}, \\
\alpha_x \geq \{\alpha_p \geq () , \alpha_p \geq \text{'False } \alpha_p \} \rightarrow \{\alpha_r \geq \text{int}\}, \\
\alpha_x \geq \alpha_x \alpha_x \alpha_x, \\
\alpha_x \geq \alpha_x \alpha_x \alpha_x \\
\alpha_x \geq + \alpha_x \alpha_x
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq \text{'True ...}, \quad \alpha_b \geq \text{'False ...}, \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq () \}, \alpha_{p2} \geq \text{'True} \alpha_{p1} \rightarrow \{\alpha_{r1} \geq \text{char}\}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq () \}, \alpha_{p4} \geq \text{'False} \alpha_{p3} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_{b}, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq \text{'True...}, \quad \alpha_b \geq \text{'False...}, \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq () ,\alpha_{p2} \geq \text{'True } \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{char}\}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq () ,\alpha_{p4} \geq \text{'False } \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_b, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq 'True \ldots, \quad \alpha_b \geq 'False \ldots, \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq () , \alpha_{p2} \geq 'True \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{char}\}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq () , \alpha_{p4} \geq 'False \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_b, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]

\[\alpha_{x3} \geq \text{int},\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b &\geq \text{`True ...}, \quad \alpha_b \geq \text{`False ...}, \\
\alpha_{x1} &\geq \text{int}, \\
\alpha_{x2} &\geq \text{int}, \\
\alpha_{x3} &\geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} &\geq \{\alpha_{p1} \geq ()}, \alpha_{p2} \geq \text{'True } \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{char}\}, \\
\alpha_{x5} &\geq \{\alpha_{p3} \geq ()}, \alpha_{p4} \geq \text{'False } \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} &\geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} &\geq \alpha_{x6} \alpha_b, \\
\alpha_{x8} &\geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq 'True \ldots, \quad \alpha_b \geq 'False \ldots, \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq (), \alpha_{p2} \geq 'True \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{char}\}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq (), \alpha_{p4} \geq 'False \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} & \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_{b}, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[\begin{align*}
\alpha_b & \geq 'True \ldots', & \alpha_b & \geq 'False \ldots', \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq ()\}, \alpha_{p2} \geq 'True \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{char}\}, & \alpha_{x3} & \geq \text{int}, & \alpha_{r1} & \geq \text{char}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq ()\}, \alpha_{p4} \geq 'False \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_b, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{align*}\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq 'True \ldots, \quad \alpha_b \geq 'False \ldots, \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq () \}, \alpha_{p2} \geq 'True \alpha_{p1} \} \rightarrow \{\alpha_{r1} \geq \text{char}\}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq () \}, \alpha_{p4} \geq 'False \alpha_{p3} \} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_b, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b &\geq 'True\ldots, \\
\alpha_{x1} &\geq \text{int}, \\
\alpha_{x2} &\geq \text{int}, \\
\alpha_{x3} &\geq +\alpha_{x1}\alpha_{x2}, \\
\alpha_{x4} &\geq \{\alpha_{p1} \geq ()\}, \alpha_{p2} \geq 'True\alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{char}\}, \\
\alpha_{x5} &\geq \{\alpha_{p3} \geq ()\}, \alpha_{p4} \geq 'False\alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} &\geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} &\geq \alpha_{x6}\alpha_{b}, \\
\alpha_{x8} &\geq +\alpha_{x3}\alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq 'True \ldots, \quad \alpha_b \geq 'False \ldots, \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq () , \alpha_{p2} \geq 'True \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{char}\}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq () , \alpha_{p4} \geq 'False \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_b, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b & \geq \text{'True ...}, \quad \alpha_b \geq \text{'False ...}, \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{ \alpha_{p1} \geq () , \alpha_{p2} \geq \text{'True } \alpha_{p1} \} \rightarrow \{ \alpha_{r1} \geq \text{char} \}, \\
\alpha_{x5} & \geq \{ \alpha_{p3} \geq () , \alpha_{p4} \geq \text{'False } \alpha_{p3} \} \rightarrow \{ \alpha_{r2} \geq \text{int} \}, \\
\alpha_{x6} & \geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_b, \quad \alpha_{x7} \geq \text{char}, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{align*}
\alpha_b &\geq \text{'True ...'}, \quad \alpha_b \geq \text{'False ...'}, \\
\alpha_{x1} &\geq \text{int}, \\
\alpha_{x2} &\geq \text{int}, \\
\alpha_{x3} &\geq + \alpha_{x1} \alpha_{x2}; \\
\alpha_{x4} &\geq \{\alpha_{p1} \geq ()\}, \alpha_{p2} \geq \text{'True} \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{char}\}, \\
\alpha_{x5} &\geq \{\alpha_{p3} \geq ()\}, \alpha_{p4} \geq \text{'False} \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} &\geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} &\geq \alpha_{x6} \alpha_b, \quad \alpha_{x7} \geq \text{char}, \quad \alpha_{x7} \geq \text{int}, \\
\alpha_{x8} &\geq + \alpha_{x3} \alpha_{x7}
\end{align*}
\]
Perform Constraint Closure

\[
\begin{aligned}
\alpha_b &\geq 'True \ldots, \\
\alpha_b &\geq 'False \ldots, \\
\alpha_{x1} &\geq \text{int}, \\
\alpha_{x2} &\geq \text{int}, \\
\alpha_{x3} &\geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} &\geq \{\alpha_{p1} \geq () , \alpha_{p2} \geq 'True \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{char}\}, \\
\alpha_{x5} &\geq \{\alpha_{p3} \geq () , \alpha_{p4} \geq 'False \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} &\geq \alpha_{x4} & \alpha_{x5}, \\
\alpha_{x7} &\geq \alpha_{x6} \alpha_b, \quad \alpha_{x7} \geq \text{char}, \quad \alpha_{x7} \geq \text{int}, \\
\alpha_{x8} &\geq + \alpha_{x3} \alpha_{x7}
\end{aligned}
\]
Perform Constraint Closure

\[
\begin{aligned}
\alpha_b & \geq 'True \ldots', \quad \alpha_b \geq 'False \ldots', \\
\alpha_{x1} & \geq \text{int}, \\
\alpha_{x2} & \geq \text{int}, \\
\alpha_{x3} & \geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq ()\}, \alpha_{p2} \geq 'True \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{char}\}, \quad \alpha_{r1}, \geq \text{char}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq ()\}, \alpha_{p4} \geq 'False \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \quad \alpha_{r2}, \geq \text{int}, \\
\alpha_{x6} & \geq \alpha_{x4} & \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \alpha_b, \quad \alpha_{x7} \geq \text{char}, \quad \alpha_{x7} \geq \text{int}, \\
\alpha_{x8} & \geq + \alpha_{x3} \alpha_{x7}, \\
\alpha_{x3} & \geq \text{int},
\end{aligned}
\]
Check Consistency

\[
\begin{align*}
\alpha_b & \geq \text{‘True …, } \quad \alpha_b \geq \text{‘False …,} \\
\alpha_{x1} & \geq \text{int,} \\
\alpha_{x2} & \geq \text{int,} \\
\alpha_{x3} & \geq + \alpha_{x1} \; \alpha_{x2}; \\
\alpha_{x4} & \geq \{\alpha_{p1} \geq () \}, \alpha_{p2} \geq \text{‘True } \alpha_{p1} \rightarrow \{\alpha_{r1} \geq \text{char}\}, \\
\alpha_{x5} & \geq \{\alpha_{p3} \geq () \}, \alpha_{p4} \geq \text{‘False } \alpha_{p3} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} & \geq \alpha_{x4} \; \& \; \alpha_{x5}, \\
\alpha_{x7} & \geq \alpha_{x6} \; \alpha_b, \quad \alpha_{x7} \geq \text{char}, \quad \alpha_{x7} \geq \text{int}, \\
\alpha_{x8} & \geq + \alpha_{x3} \; \alpha_{x7}, \\
\end{align*}
\]
Check Consistency

\[
\begin{align*}
\alpha_b &\geq 'True \ldots, \quad \alpha_b \geq 'False \ldots, \\
\alpha_{x1} &\geq \text{int}, \\
\alpha_{x2} &\geq \text{int}, \\
\alpha_{x3} &\geq + \alpha_{x1} \alpha_{x2}, \\
\alpha_{x4} &\geq \{\alpha_{p1} \geq () , \alpha_{p2} \geq 'True \alpha_{p1}\} \rightarrow \{\alpha_{r1} \geq \text{char}\}, \\
\alpha_{x5} &\geq \{\alpha_{p3} \geq () , \alpha_{p4} \geq 'False \alpha_{p3}\} \rightarrow \{\alpha_{r2} \geq \text{int}\}, \\
\alpha_{x6} &\geq \alpha_{x4} \& \alpha_{x5}, \\
\alpha_{x7} &\geq \alpha_{x6} + \alpha_{b}, \quad \alpha_{x7} \geq \text{char}, \quad \alpha_{x7} \geq \text{int}, \\
\alpha_{x8} &\geq + \alpha_{x3} \alpha_{x7},
\end{align*}
\]

\[\times\]
Typechecking by Example

First, we will typecheck this program:

1. `let b = ...`
2. `1 + 2 + (if b then 5 else 1)`

Then, we will typecheck this program:

1. `let b = ...`
2. `1 + 2 + (if b then 'z' else 1)`