Statically Identifying Duck Typing

Duck Typing Analysis

Michael Jungo

24th May 2016

SC: Software Composition Seminar

Duck Typing

What is duck typing?

If it quacks like a duck and walks like a duck, it is a duck.

- Used in dynamic programming languages
- Suitability by presence of properties or methods
- Runtime errors if incompatible
- No guarantee that it behaves correctly

Example

```ruby
class Duck
  def quack
    puts 'quack'
  end
end

class Human
  def quack
    puts 'I am a duck'
  end
end
```

```ruby
def make_sound(duck)
  duck.quack
end

duck = Duck.new
human = Human.new

make_sound(duck)
make_sound(human)
```

Name ambiguity

Method names are not sufficient to determine duck typing.

- Different number of arguments
- Completely unrelated

  `GLMCheckboxBrick>>#select:`

  `Dictionary>>#select:`

Cartesian Product Algorithm

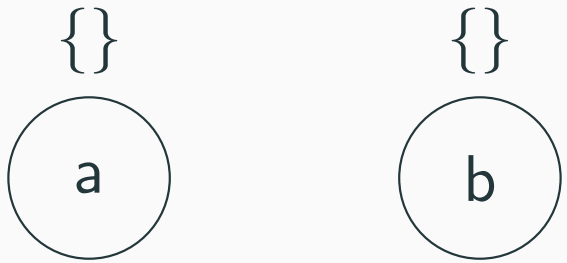
Cartesian Product Algorithm

- Type inference for variables
- Static analysis
- Originally developed for the language Self by Agesen
- Builds a graph where:
  - Nodes represent types of a variable
  - Directed edges represent constraints
- Consists of three steps

Step 1 - Allocating type variables

Create an empty node for each variable.

```
a = 'some string'
b = 2.5
b = a
```



Step 2 - Seeding type variables

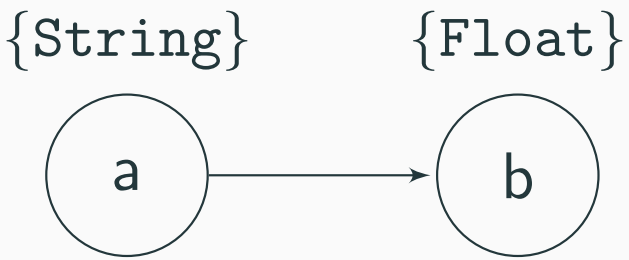
Set initial type of variables.

```
a = 'some string'
b = 2.5
b = a
```

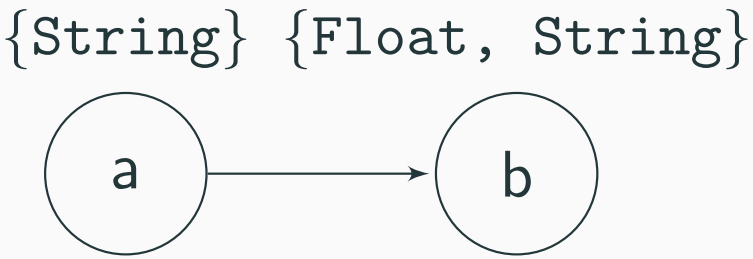
Step 3 - Constraints and propagation

```
a = 'some string'
b = 2.5
b = a
```

Add constraints (edges) to represent data flow.



Propagate types along the edges.



Data flow and polymorphism

Flow sensitive

Preserves only the last assigned type.

Flow insensitive

Keeps a set of all assigned types.

CPA is flow insensitive

Data polymorphism

Variables can hold different types.

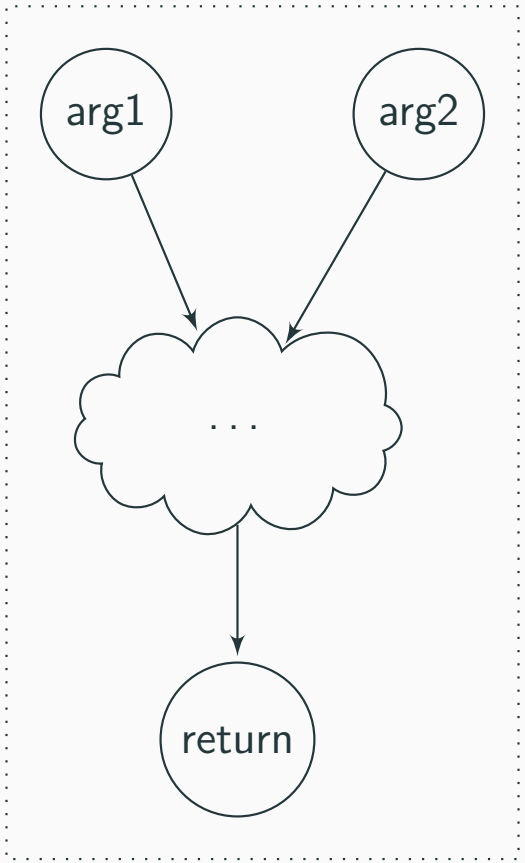
Parametric polymorphism

Methods accept multiple types as input parameters.

CPA handles this with method templates

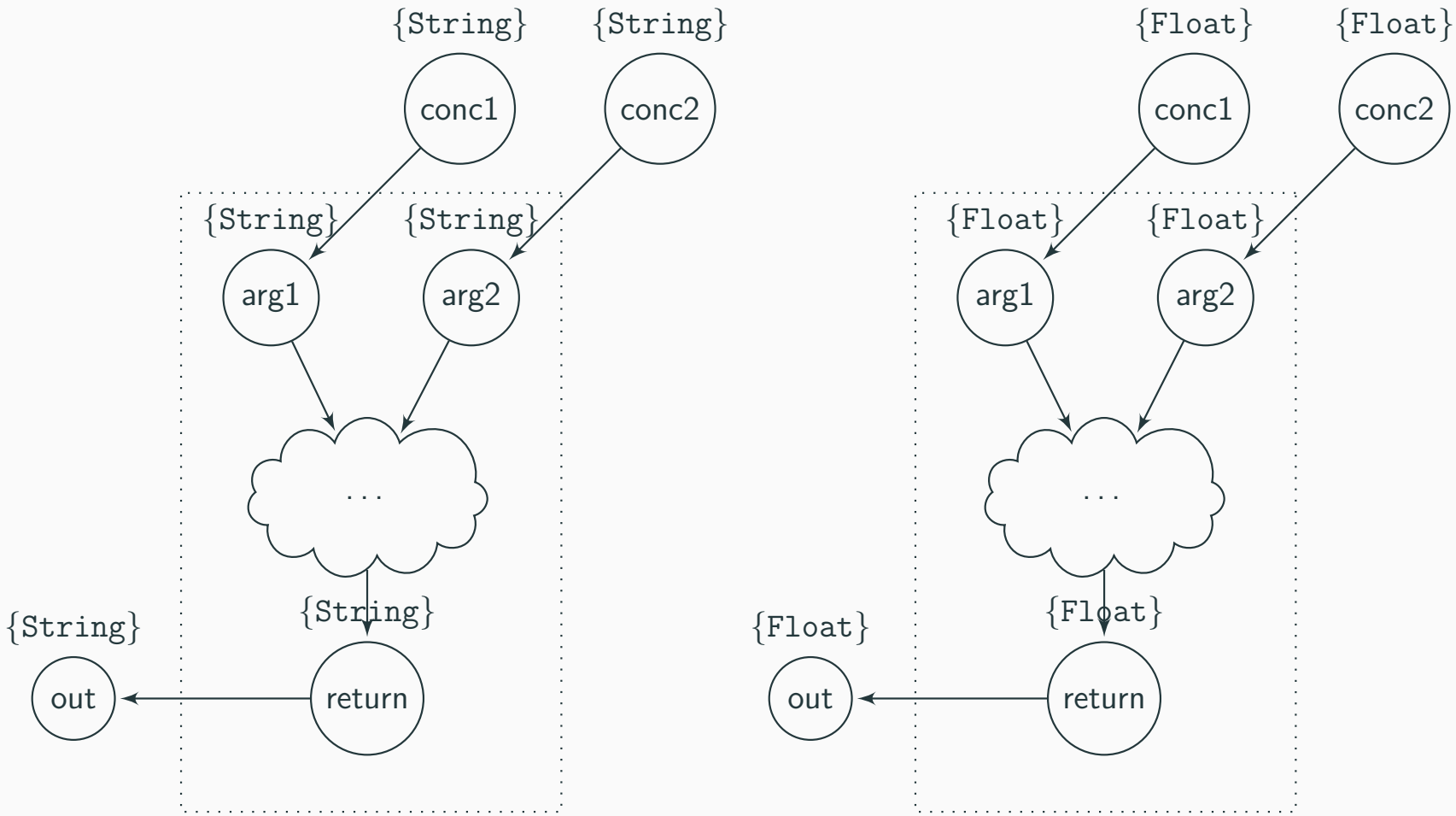
Method templates

A subgraph representing the method's body.



Concrete method calls

Copy the template and apply the concrete argument types.



CPA to find duck types

Modifications to CPA

1. Consider variables that call a method.
2. Collect all types of the variable that respond to the method.
3. Traverse the said method of these types.
4. Types that successfully traversed the method are possible candidates for duck typing.
5. Continue until the algorithm stops.

Are the remaining candidates really duck typed?

There is no guarantee.

- Likelihood increases with the number of method calls
- Types within the same hierarchy must be excluded
- Unrelated methods might still not be eliminated
- Further analysis can be done
- User could decide

Problems

- Blocks are difficult to analyse
- Primitives need to be handled
- Small parts are imprecise
- Cannot be initiated at any desired point (missing information)

What could it be used for?

Refactoring

Provide informations about potential incompatibility due to changes.

Unexpected call

Detect types that were not supposed to reach a certain point.

Curiosity

Debatable duck typing

```
list = [duck, human]

list.each do |elem|
  elem.quack
end
```

```
var = duck
var.quack

var = human
var.quack
```

Is elem duck typed?

How about var?