Unification

in Ruby

Martin Kay
Stanford University
University of the Saarland

This program is intended to provide a broad basis for unification in Ruby. It depends crucially on a new class of objects called

Variables

which can have

Values

Hash tables are made to mimic attribute values to the extent possible, and arrays mimic Prolog terms.

A compromise

After unification, hash tables may be left with unbound variables in some positions that were not there when the process began.


```
% v = unify(7, 7)
=> #<struct BoundValue value=7>
% v.value
=> 7
% v = unify('b', 'a')
=> nil
% v = unify(nil, nil)
=> #<struct BoundValue value=nil>
% v.value
=> nil
% v = unify(false, nil)
=> nil
```

```
% unify([1,2,3], [1,2,3]).value
=> [1, 2, 3]
% unify([1,'a',true], [1,'a',true]).value
=> [1, "a", true]
% unify([1,'a',true], [1,'a'])
=> nil
% unify({'a' => 1, 'b' => 2},
        {'a' => 1, 'b' => 2}).value
=> {"a"=>1, "b"=>2}
% unify({'a' => 1, 'b' => 2},
        {'a' => 1, 'c' => 3}).value
=> {"a"=>1, "b"=>2, "c"=>3}
% unify({'a' => 1, 'b' => 2}, {'a' => 3, 'c' => 2})
=> nil
```



```
% a = {'cat' => 'adj',
       'agr' => {'num' => 'sg',
                 'gender' => 'fem'}}
=> ...
% b = {'agr' => {'gender' => 'fem',
                 'case' => 'nom'}}
=> ...
% v = unify(a, b)
=> #<struct BoundValue
      value={"cat"=>"adj",
             "agr"=>{"num"=>"sg",
                     "gender"=>"fem",
                     "case"=>"nom"}}>
```



```
% v = Variable.new
=> #<Variable:0x38fff4 @status=nil, @value=nil>
% b = BindingStatus.new(true)
=> #<BindingStatus:0x38a388 @bound=true>
% v.bind(b, 12)
=> 12
% v.last_binding
=> 12
% b.reset
=> false
% v.last_binding
=> #<Variable:0x38fff4 @status=#<BindingStatus:
0x38a388 @bound=false>, @value=12>
```


How does it work?

```
def unifiable
  false
end

def normalize
  self
end

def bound
  true
end

def binding
  self
end
```

Define some simple methods for all objects unless specifically overridden.

```
def unifiable
  false
end

def bound
  true
end

def first_binding
  self
end

def last_binding
  self
end
```

Most kinds of things can only be equal or unequal. Only arrays and hashes are 'unifiable' at the moment.


```
def unifiable
  false
end

def bound
  true
end

def first_binding
  self
end

def last_binding
  self
end
```

Only variables can be unbound

Classes

```
BoundValue = Struct.new(:value)
```

If unification succeeds, wrap the result up in a 'BoundValue'. Then we will recognize failure because it will be just plain old nil

```
class BindingStatus
```

All new variables created during unification point to the same valid BindingStatus. If unification fails, we make this invalid, and all the variables become unbound.

```
class Variable
    @status=nil
    @value=nil
```


Classes

```
class Variable

  def initialize
  def to_s
  def get_name
  def show
```

...for display

```
  def copy
  def bind(status, value)
  def bound
  def last_binding
  def first_binding
  def occurs_within(exp)
  attr_accessor :status
end
```

Classes

```
class Variable

  def initialize
  def to_s
  def get_name
  def show

  def copy
  def bind(status, value)
  def bound
  def last_binding
  def first_binding
  def occurs_within(exp)
  attr_accessor :status
end
```

...for the algorithm

BindingStatus

```
class BindingStatus

  def initialize(s = true)
    @bound=s
  end

  def bound
    @bound ? true : false
  end

  def reset
    @bound=false
  end

  def to_s
    @bound ? 'bound' : 'unbound'
  end

end
```

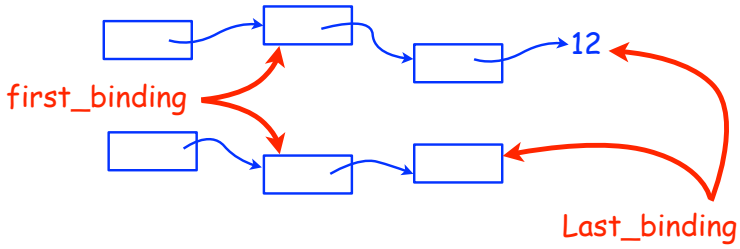

Variable: bind and bound

```
def bind(status, value)
  @status=status
  @value=value
end

def bound
  if @status.class==BindingStatus
    @status.bound
  else
     @status
  end
end
```

Status can be just true or false

Variable: bind and bound

```
def last_binding
  bound ? @value.last_binding : self
end

def first_binding
  bound ? @value : self
end
```



Make Arrays and Hashes look alike

```
class Array

  def unifiable
    true
  end

  def each_key
    length.times {|i|
  end

  def each_pair
    each_key {|i| yiel
  end

  def each_value
    each{|i| yield i}
  end

  def get_item(i)
    self[i]
  end

end
```

```
class Hash

  def unifiable
    true
  end

  def get_item(i)
    return self[i] if has_key?(i)
    self[i]=Variable.new
  end

end
```


The "occurs check"

```
def occurs_within(exp)
  return true if self==exp
  if exp.unifiable
    exp.each_value do |e|
      return true if occurs_within(e)
    end
  end
  false
end
```

Outlaw cyclic structures

The unifier

```
def unify_exprs(exp1, exp2, level=nil, status=nil)
  status=BindingStatus.new(true) if not status
  val1=exp1.last_binding
  val2=exp2.last_binding
  return status if val1==val2

    val1 is a variable

    val2 is a variable

    val1 and val2 are "unifiable"

  end
```

The simple case

Identical up to renaming

The unifier

```
Val1 is a variable

if val1.is_a?(Variable)
    if val1.occurs_within(val2)
      status.reset()
      return false
    end
    val1.bind(status, val2)
    return status
  end
```

Return the status. Reset it to show failure

Bind the other value to it with the current status

```
if val1.unifiable and val1.class==val2.class
    val1.each_key do |i|
      v1=val1.get_item(i)
      if (v2=val2.get_item(i)).bound
        if not unify_exprs(v1, v2, level, status)
          status.reset()
          return status
        end
      else
        v2.bind(status, v1)
      end
    end
    val2.each_key do |i|
        unless (v1=val1.get_item(i)).bound
          v1.bind(status, val2.get_item(i))
        end
    end
    return status
  end
end
```

Same attribute


```
if val1.unifiable and val1.class==val2.class
    val1.each_key do |i|
      v1=val1.get_item(i)
      if (v2=val2.get_item(i)).bound
        if not unify_exprs(v1, v2, level, status)
          status.reset()
          return status
        end
      else
        v2.bind(status, v1)
      end
    end
    val2.each_key do |i|
        unless (v1=val1.get_item(i)).bound
          v1.bind(status, val2.get_item(i))
        end
    end
    return status
  end
end
```

Fail if recursive unify fails


```
if val1.unifiable and val1.class==val2.class
    val1.each_key do |i|
      v1=val1.get_item(i)
      if (v2=val2.get_item(i)).bound
        if not unify_exprs(v1, v2, level, status)
          status.reset()
          return status
        end
      else
        v2.bind(status, v1)
      end
    end
    val2.each_key do |i|
        unless (v1=val1.get_item(i)).bound
          v1.bind(status, val2.get_item(i))
        end
    end
    return status
  end
end
```

If v2 is unbound, bind v1 to it


```
if val1.unifiable and val1.class==val2.class
    val1.each_key do |i|
      v1=val1.get_item(i)
      if (v2=val2.get_item(i)).bound
        if not unify_exprs(v1, v2, level, status)
          status.reset()
          return status
        end
      else
        v2.bind(status, v1)
      end
    end
    val2.each_key do |i|
        unless (v1=val1.get_item(i)).bound
          v1.bind(status, val2.get_item(i))
        end
    end
    return status
  end
end
```

Check the attributes in val2 that are not in val1

copy

```
def copy

  val = last_binding
  if val.unifiable
    cp = val.class.new
    val.each_pair do |a, v|
      cp[a] = v.copy
    end
    return cp
  end
  val

end
```


unify

```
BoundValue = Struct.new(:value)

def unify(a, b, level=nil)
  s=unify_exprs(a, b, level)
  if s && s.bound
    c=a.copy
    s.reset()
    return BoundValue.new(c)
  end
  return nil
end
```

Success!

unify

```
BoundValue = Struct.new(:value)

def unify(a, b, level=nil)
  s=unify_exprs(a, b, level)
  if s && s.bound
    c=a.copy
    s.reset()
    return BoundValue.new(c)
  end
  return nil
end
```

Failure