Constraint Programming

Thursday, February 26

Motivation

```
9 * celsius = 5 * (fahrenheit - 32)

Given celsius, how do you find fahrenheit?


Given fahrenheit, how do you find celsius?
```

Motivation

```
9 * celsius = 5 * (fahrenheit - 32)

Given celsius, how do you find fahrenheit?
fahrenheit = (9 * celsius) / 5 + 32

Given fahrenheit, how do you find celsius?
celsius = 5 * (fahrenheit - 32) / 9
```

Declarative vs Procedural

Declarative: describes how different quantities relate to one another (multi-directional)

Procedural: describes how to compute a particular result from a particular set of inputs (one-directional)

Declarative vs Procedural

Algebraic equations are declarative

```
9 * celsius = 5 * (fahrenheit - 32)
```

Python functions are procedural

```
def get_celsius(fahrenheit):
   return 5 * (fahrenheit - 32) / 9

def get_fahrenheit(celsius):
   return (9 * celsius) / 5 + 32
```

Declarative Programming

Constraint programming is declarative programming

- Define relationships between quantities
- Provide values for knowns
- System computes values for unknowns

Defining Simple Relationships

Constraints enforce mathematical relationships

E.g.: `multiplier(celsius, w, u)` enforces the mathematical relationship `celsius * w = u`

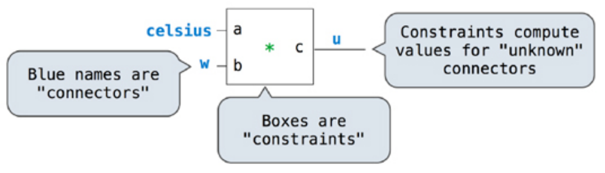
Defining Complex Relationships

Connectors combine simple constraints to describe complex ones

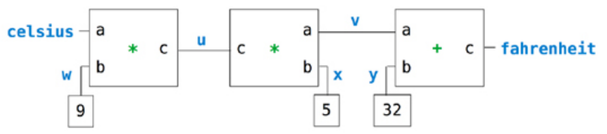
A connector holds a value and is a parameter to one or more constraints

“Complex relationship” == “constraint network”

A Simple Relationship, Visually


celsius
w
a
b
*
c
u
Blue names are "connectors"
Constraints compute values for "unknown" connectors
Boxes are "constraints"

A Complex Relationship, Visually



Implementing Constraint Networks

Connectors need to keep track of their values and know when values they care about change.

How can we do this?

Message Passing

Pieces of a program communicate by passing messages to each other:

- What do you have for question 3?
- Change your answer for question 5 to C.

Implementing Message Passing

One function encapsulates the behavior of all operations on a piece of data and responds to different messages.

How can we do this?

A Bunch of Conditionals

We could use conditionals to check the message, then perform the appropriate function.

But:

- Equality tests are repetitive
- We can’t add new messages without adding more conditionals

Can we do better?

Dispatch Dictionaries

Dictionaries handle the message look-up logic, so we can focus on implementing useful behavior.

```
dispatch_dictionary = {
   'message_1': function_or_data_object_1
   'message_2': function_or_data_object_2
}
```

Using Dispatch Dictionaries

How can dispatch dictionaries help us build a constraint network?

What objects in our network can we represent with dispatch dictionaries?

Constraint's Interface

```
constraint = some_constraint(a, b, c)
```

`constraint['new_val'](source, value)`: `source` (a connector) has a new value, `value`.

`constraint['forget'](source)`: `source` (a connector) has forgotten `source`'s value.

Connector's Interface

```
connector = make_connector('Celsius')
```

`connector['set_val'](source, value)`: `source` (a string identifier or a constraint) tells `connector` to set `connector`'s value to `value`.

`connector['has_val']()`: does `connector` already have a value?

`connector['val']`: returns `connector`'s value.

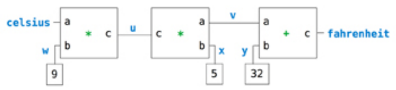
`connector['forget'](source)`: `connector` forgets `connector`'s value if `source` (a string identifier or a constraint) is `connector`'s informant.

`connector['connect'](source)`: `source` (a constraint) tells `connector` to be one of `source`'s parameters.

A Complex Relationship in Code

```
def make_converter(celsius, fahrenheit):
    u, v, w, x, y = [make_connector() for _ in range(5)]
    multiplier(celsius, w, u)
    multiplier(v, x, u)
    adder(v, y, fahrenheit)
    constant(w, 9)
    constant(x, 5)
    constant(y, 32)

celsius = make_connector('Celsius')
fahrenheit = make_connector('Fahrenheit')
make_converter(celsius, fahrenheit)
```



Demo

Let's put our converter to work!

Implementing an Adder Constraint

```
def adder_constraint(a, b, c):
    def new_value():
        pass # Will implement soon

    def forget_value():
        for connector in (a, b, c):
            connector['forget'](constraint)

    constraint = {'new_val': new_value, 'forget': forget_value}

    for connector in (a, b, c):
        connector['connect'](constraint)

    return constraint
```

Building a Generic Constraint

```
def make_ternary_constraint(a, b, c, ab, ca, cb):
    """The constraint that ab(a,b)=c and ca(c,a)=b and cb(c,b)=a."""
    def new_value():
        av, bv, cv = [connector['has_val']() for connector in (a, b, c)]
        if av and bv:
            c['set_val'](constraint, ab(a['val'], b['val']))
        elif av and cv:
            b['set_val'](constraint, ac(c['val'], a['val']))
        elif bv and cv:
            a['set_val'](constraint, cb(c['val'], b['val']))

    """Rest of function is same as adder_constraint"""
```

Making ternary constraints

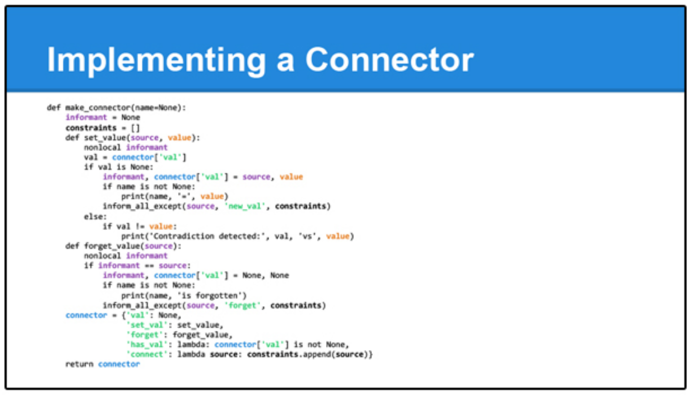
```
from operator import add, sub, mul, truediv

def adder(a, b, c):
    """The constraint that a + b = c."""
    return make_ternary_constraint(a, b, c, add, sub, sub)

def multiplier(a, b, c):
    """The constraint that a * b = c."""
    return make_ternary_constraint(a, b, c, mul, truediv, truediv)
```

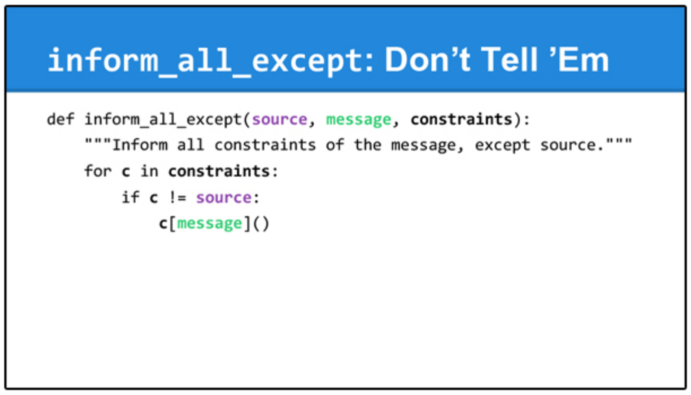
Constant Constraint: Never Forget

```
def constant(connector, value):
    """The constraint that connector = value."""
    constraint = {}
    connector['set_val'](constraint, value)
    return constraint
```



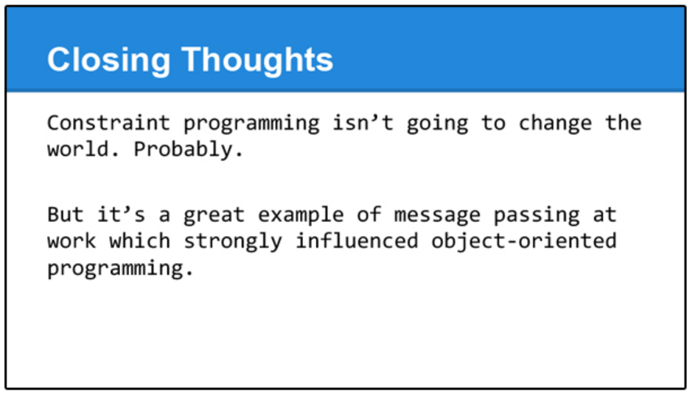
Implementing a Connector

```python
def make_connector(name=None):
    informant = None
    constraints = []
    def set_value(source, value):
        nonlocal informant
        val = connector['val']
        if val is None:
            informant, connector['val'] = source, value
            if name is not None:
                print(name, '=', value)
            inform_all_except(source, 'new_val', constraints)
        else:
            if val != value:
                print('Contradiction detected:', val, 'vs', value)
    def forget_value(source):
        nonlocal informant
        if informant == source:
            informant, connector['val'] = None, None
            if name is not None:
                print(name, 'is forgotten')
            inform_all_except(source, 'forget', constraints)
    connector = {'val': None,
                 'set_val': set_value,
                 'forget': forget_value,
                 'has_val': lambda: connector['val'] is not None,
                 'connect': lambda source: constraints.append(source)}
    return connector
```



inform_all_except: Don't Tell 'Em

```python
def inform_all_except(source, message, constraints):
    """Inform all constraints of the message, except source."""
    for c in constraints:
        if c != source:
            c[message]()
```



Closing Thoughts

Constraint programming isn't going to change the world. Probably.

But it's a great example of message passing at work which strongly influenced object-oriented programming.