61A LECTURE 26 – UNIFICATION, HALTING PROBLEM

Steven Tang and Eric Tzeng
August 7, 2013

Announcements

- Final exam review session this weekend
  - Friday 1-5 pm, room TBA
  - See Piazza Poll to vote on additional times
  - Potential extra credit – more information later in the week

Logic Language Review

Expressions begin with *query* or *fact* followed by relations

Expressions and their relations are Scheme lists

```
logic> (fact (parent eisenhower fillmore))
logic> (fact (parent fillmore abraham))
logic> (fact (parent abraham clinton))
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y))
logic> (query (ancestor ?who abraham))
Success!
who: fillmore
who: eisenhower
```

If a fact has more than one relation, the first is the *conclusion*, and it is satisfied if the remaining relations, the *hypotheses*, are satisfied

If a query has more than one relation, all must be satisfied

The interpreter lists all bindings that it can find to satisfy the query

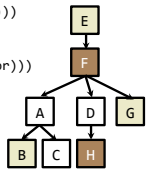
Hierarchical Facts

Relations can contain relations in addition to atoms

```
logic> (fact (dog (name abraham) (color white)))
logic> (fact (dog (name barack) (color tan)))
logic> (fact (dog (name clinton) (color white)))
logic> (fact (dog (name delano) (color white)))
logic> (fact (dog (name eisenhower) (color tan)))
logic> (fact (dog (name fillmore) (color brown)))
logic> (fact (dog (name grover) (color tan)))
logic> (fact (dog (name herbert) (color brown)))
```

Variables can refer to atoms or relations

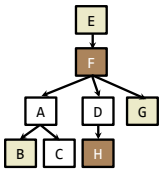
```
logic> (query (dog (name clinton) (color ?color)))
Success!
color: white

logic> (query (dog (name clinton) ?info))
Success!
info: (color white)
```



Example: Combining Multiple Data Sources

Which dogs have an ancestor of the same color?

```
logic> (query (dog (name ?name) (color ?color))
              (ancestor ?ancestor ?name)
              (dog (name ?ancestor) (color ?color)))
Success!
name: barack    color: tan     ancestor: eisenhower
name: clinton   color: white   ancestor: abraham
name: grover    color: tan     ancestor: eisenhower
name: herbert   color: brown   ancestor: fillmore
```



Example: Appending Lists

Two lists append to form a third list if:

- The first list is empty and the second and third are the same

  `() (a b c) (a b c)`

- Both of the following hold:
  - List 1 and 3 have the same first element
  - The rest of list 1 and all of list 2 append to form the rest of list 3

  `(a b c) (d e f) (a b c d e f)`

```
logic> (fact (append-to-form () ?x ?x))

logic> (fact (append-to-form (?a . ?r) ?y (?a . ?z))
             (append-to-form ?r ?y ?z))
```

Pattern Matching

The basic operation of the Logic interpreter is to attempt to unify two relations

Unification is finding an assignment to variables that makes two relations the same

```
( (a  b) c  (a  b) )
(   ?x   c    ?x   )
```

▷ True, {x: (a b)}

```
( (a  b) c  (a  b) )
( (a ?y) ?z (a  b) )
```

▷ True, {y: b, z: c}

```
( (a  b) c  (a  b) )
(   ?x  ?x    ?x   )
```

▷ False

Unification

Unification unifies each pair of corresponding elements in two relations, accumulating an assignment

1. Look up variables in the current environment
2. Establish new bindings to unify elements

```
( (a  b) c (a  b) )
(   ?x   c   ?x   )
```

Lookup

```
(a  b)
(a  b)
```

{ x: (a b) }

Success!

```
( (a  b) c  (a  b) )
(   ?x  ?x    ?x   )
```

Lookup

```
c
(a b)
```

{ x: (a b) }

Failure.

Symbols/relations without variables only unify if they are the same

Unification with Two Variables

Two relations that contain variables can be unified as well

```
(   ?x      ?x     )
( (a ?y c) (a b ?z) )
```

▷ True, { x: (a ?y c),
y: b,
z: c}

Lookup

```
(a ?y c)
(a  b ?z)
```

Substituting values for variables may require multiple steps

`lookup('?x')` ⇨ `(a ?y c)` `lookup('?y')` ⇨ `b`

Implementing Unification

```
def unify(e, f, env):
    e = lookup(e, env)
    f = lookup(f, env)
    if e == f:
        return True
    elif isvar(e):
        env.define(e, f)
        return True
    elif isvar(f):
        env.define(f, e)
        return True
    elif scheme_atomp(e) or scheme_atomp(f):
        return False
    else:
        return unify(e.first,  f.first,  env) and \
               unify(e.second, f.second, env)
```

1. Look up variables in the current environment

Symbols/relations without variables only unify if they are the same

2. Establish new bindings to unify elements.

Unification recursively unifies each pair of elements

Searching for Proofs

The Logic interpreter searches the space of facts to find unifying facts and an env that prove the query to be true

```
(fact  (app () ?x ?x))
(fact  (app (?a . ?r) ?y (?a . ?z))
       (app      ?r  ?y      ?z ))
(query (app ?left (c d) (e b c d)))
```

```
(app ?left (c d) (e b c d))
    {a: e, y: (c d), z: (b c d), left: (?a . ?r)}
    (app (?a . ?r) ?y (?a . ?z))
        conclusion <- hypothesis
    (app ?r (c d) (b c d)))
    {a2: b, y2: (c d), z2: (c d), r: (?a2 . ?r2)}
    (app (?a2 . ?r2) ?y2 (?a2 . ?z2))
        conclusion <- hypothesis
    (app ?r2 (c d) (c d))
    {r2: (), x: (c d)}
    (app () ?x ?x)
```

Variables are local to facts and queries

Left: (e . (b . ())) ⇨ (e b)

Underspecified Queries

Now that we know about Unification, let's look at an underspecified query

What are the results of these queries?

```
> (fact (append-to-form () ?x ?x))

> (fact (append-to-form (?a . ?r) ?x (?a . ?s))
        (append-to-form ?r ?x ?s))

> (query (append-to-form (1 2) (3) ?what))
Success!
what: (1 2 3)

> (query (append-to-form (1 2 . ?r) (3) ?what)
Success!
r: ()   what: (1 2 3)
r: (?s_6)      what: (1 2 ?s_6 3)
r: (?s_6 ?s_8) what: (1 2 ?s_6 ?s_8 3)
r: (?s_6 ?s_8 ?s_10)  what: (1 2 ?s_6 ?s_8 ?s_10 3)
r: (?s_6 ?s_8 ?s_10 ?s_12)   what: (1 2 ?s_6 ?s_8 ?s_10 ?s_12 3)
...
```

Search for possible unification

The space of facts is searched exhaustively, starting from the query and following a *depth-first* exploration order

A possible proof is explored exhaustively before another one is considered

```
def search(clauses, env):
  for fact in facts:
    env_head <- unify(conclusion of fact, first clause, env)
    if unification succeeds:
        env_rule <- search(hypotheses of fact, env_head)
            result <- search(rest of clauses, env_rule)
              yield each result
```

Some good ideas:

- Limiting depth of the search avoids infinite loops
- Each time a fact is used, its variables are renamed
- Bindings are stored in separate frames to allow backtracking

Implementing Search

```
def search(clauses, env, depth):
  if clauses is nil:
    yield env
  elif DEPTH_LIMIT is None or depth <= DEPTH_LIMIT:
    for fact in facts:
      fact = rename_variables(fact, get_unique_id())
      env_head = Frame(env)
      if unify(fact.first, clauses.first, env_head):
        for env_rule in search(fact.second, env_head, depth+1):
          for result in search(clauses.second, env_rule, depth+1):
            yield result
```

Whatever calls search can access all yielded results

An Evaluator in Logic

We can define an evaluator in Logic; first, we define numbers:

```
logic> (fact (ints 1 2))
logic> (fact (ints 2 3))
logic> (fact (ints 3 4))
logic> (fact (ints 4 5))
```

Then we define addition:

```
logic> (fact (add 1 ?x ?y) (ints ?x ?y))
logic> (fact (add ?x ?y ?z)
             (ints ?x-1 ?x) (ints ?z-1 ?z) (add ?x-1 ?y ?z-1))
```

Finally, we define the evaluator:

```
logic> (fact (eval ?x ?x) (ints ?x ?something))
logic> (fact (eval (+ ?op0 ?op1) ?val)
             (add ?a0 ?a1 ?val) (eval ?op0 ?a0) (eval ?op1 ?a1))
logic> (query (eval (+ 1 (+ ?what 2)) 5))
Success!
what: 2
what: (+ 1 1)
```

The Halting Problem

Robert Huang

August 7, 2013

Review: Program Generator

A computer program is just a sequence of bits

It is possible to enumerate all bit sequences

```
from itertools import product

def bitstrings():
    size = 0
    while True:
        tuples = product(('0', '1'), repeat=size)
        for elem in tuples:
            yield ''.join(elem)
        size += 1

>>> [next(bs) for _ in range(0, 10)]
['', '0', '1', '00', '01', '10', '11', '000', '001', '010']
```

Function Streams

Given a stream of 1-argument functions, we can construct a function that is not in the stream, *assuming that all functions in the stream terminate*

```
def func_not_in_stream(s):
    return lambda n: not s[n](n)
```

Inputs

```
[F] T  T  T  T  F  T  F  T  F  . . .
 T [T] F  F  F  F  F  T  F  T  . . .
 T  F [T] F  T  F  T  F  T  T  . . .
 T  F  F [T] T  F  F  T  F  T  . . .
 T  F  T  T [F] T  F  T  F  T  . . .
 F  F  F  F  T [F] F  F  T  T  . . .
 T  F  T  F  F  F [F] T  T  T  . . .
 F  T  F  T  T  F  T [F] F  T  . . .
 T  F  T  F  F  T  T  F [F] T  . . .
 F  T  T  T  T  T  T  T  T [F] . . .
                . . .
 T  F  F  F  T  T  T  T  T  T  . . .
```

Functions

Programs and Mathematical Functions

A mathematical function *f(x)* maps elements from its input *domain D* to its output *range R*

$$f : \mathbb{N} \rightarrow \{0, 1\},\ f(x) = x^2 \mod 2$$

A Python function **func** *computes* a mathematical function *f* if the following conditions hold:

- **func** has the same number of parameters as inputs to *f*
- **func** terminates on every input in *D*
- The return value of **func(x)** is the same as *f(x)* for all *x* in *D*

```
def func(x):
    return (x * x) % 2
```

A mathematical function *f* is *computable* if there exists a program (i.e. a Python function) **func** that computes it

Computability

Are all functions computable?

More specifically, we hate infinite loops

So if we have a program that computes the following function, we can run it on our programs to determine if they have infinite loops:

$$haltsonallinputs : Programs \rightarrow \{0, 1\},$$
$$haltsonallinputs(P) = \begin{cases} 1 & \text{if } P \text{ halts on all inputs} \\ 0 & \text{otherwise} \end{cases}$$

Halts

Let's be less ambitious; we'll take a program that computes whether or not another program halts on a specific non-negative integer input:

$$halts : Programs \times \mathbb{N} \rightarrow \{0, 1\},$$
$$halts(P, n) = \begin{cases} 1 & \text{if } P \text{ halts on input } n \\ 0 & \text{otherwise} \end{cases}$$

Is this function computable?

It's not as simple as just running the program *P* on *n* to see if it terminates

How long do we let it run before deciding that it won't terminate?

However long we let it run before declaring it that it won't terminate, it might just need a little more time to finish its computation

Thus, we have to do something more clever, analyzing the program itself

Turing

Let's assume that we have a Python function **halts** that computes the mathematical function *halts*, written by someone more clever than us

Remember, we can pass a function itself as its argument. Thus, we can consider **halts(f, f)**; in other words, does function **f** halt when given itself as an argument? (This is just a thought experiment.)

We can then define a new function, **turing**, which takes in 1 argument.

```
def turing(f):
    if halts(f, f):
        while True:     # infinite loop
            pass
    else:
        return True     # halts
```

turing will go into an infinite loop if **f** halts when given itself as an argument. Otherwise, **turing** returns **True**.

Turing… what?

```
def turing(f):
    if halts(f, f):
        while True:     # infinite loop
            pass
    else:
        return True     # halts

turing(turing)          # * what?
```

If this sounds fishy, it should. Should the call **turing(turing)** halt or go into an infinite loop?

- **turing(turing)** loops → **halts(turing, turing)** returns true
  - However, **turing(turing)** should have halted
- **turing(turing)** halts → **halts(turing, turing)** returns false
  - However, **turing(turing)** should not have halted

We have a contradiction! Our assumption that **halts** exists is false.

Bitstrings and Functions

Let's develop another proof, assuming that we have a **halts** program that computes the mathematical function *halts*

Let's create a stream of all 1-argument Python functions, then use **halts** to filter out non-terminating programs from that stream

Assume we have the following Python functions:

```
def is_valid_python_function(bitstring):
    """Determine whether or not a bitstring represents a
    syntactically valid 1-argument Python function."""

def bitstring_to_python_function(bitstring):
    """Coerce a bitstring representation of a Python
    function to the function itself."""
```

Bitstrings and Functions

Let's develop another proof, assuming that we have a **halts** program that computes the mathematical function *halts*

Let's create a stream of all 1-argument Python functions, then use **halts** to filter out non-terminating programs from that stream

Then the following produces all valid 1-argument Python functions:

```
def function_stream():
    """Return a stream of all valid 1-argument Python
    functions."""
                                        On HW12
    bitstring_stream = iterator_to_stream(bitstrings())
    valid_stream = filter_stream(is_valid_python_function,
                                 bitstring_stream)
    return map_stream(bitstring_to_python_function,
                      valid_stream)
```

Filtering Out Non-Terminating Programs

With **halts**, we can't filter out programs that don't halt on all input

But we can filter out programs that don't halt on a specific input

Specifically, let's make sure that a program halts on its index in the resulting stream of programs

```
def make_halt_checker():
    index = 0
    def halt_checker(fn):
        nonlocal index
        if halts(fn, index):
            index += 1
            return True
        return False
    return halt_checker

programs = filter_stream(make_halt_checker(),
                         function_stream())
```

Developing a Contradiction

We now have a stream of programs that halt when given their own index as input

```
programs = filter_stream(make_halt_checker(),
                         function_stream())
```

Recall the following function that produces a function that is not in a given stream:

```
def func_not_in_stream(s):
    return lambda n: not s[n](n)
```

Consider the following:

```
church = func_not_in_stream(programs)
```

Does **church** appear anywhere in **programs**?

Developing a Contradiction

```
def func_not_in_stream(s):
    return lambda n: not s[n](n)
```

```
church = func_not_in_stream(programs)
```

Does **church** appear anywhere in **programs**?

Every element in **programs** halts when given its own index as input

Thus, **church** halts on all inputs **n**, since it calls the **n**th element in **programs** on **n**

So **halt_checker** returns true on **church**, which means that **church** is in **programs**

If **church** is in **programs**, it has an index **m**; so what does **church(m)** do?

Developing a Contradiction

```
def func_not_in_stream(s):
    return lambda n: not s[n](n)
```

```
church = func_not_in_stream(programs)
```

Does **church** appear anywhere in **programs**?

Every element in **programs** halts when given its own index as input

Thus, **church** halts on all inputs **n**, since it calls the **n**th element in **programs** on **n**

If **church** is in **programs**, it has an index **m**; so what does **church(m)** do?

It calls the **m**th element in **programs**, which is **church** itself, on **m**

This results in an infinite loop, which means **halt_checker** will return false on **church**, since it does not halt given its own index

Developing a Contradiction

```
def func_not_in_stream(s):
    return lambda n: not s[n](n)
```

```
church = func_not_in_stream(programs)
```

We have a contradiction!

halt_checker(church) returns true, which means that **church** is in **programs**

But if **church** is in **programs**, then **church(m)**, where **m** is **church**'s index in **programs**, is an infinite loop, so **halt_checker(church)** returns false

So we made a false assumption somewhere

False Assumption

We assumed we had the following Python functions:

- **`halts`**
- **`is_valid_python_function`**
- **`bitstring_to_python_function`**

Everything else we wrote ourselves

The latter two functions can be built using components of the interpreter

Thus, it is our assumption that there is a Python function that computes *halts* that is invalid

$$halts : Programs \times \mathbb{N} \rightarrow \{0, 1\},$$
$$halts(P, n) = \begin{cases} 1 & \text{if } P \text{ halts on input } n \\ 0 & \text{otherwise} \end{cases}$$

The Halting Problem

The question of whether or not a program halts on a given input is known as *the halting problem*.

In 1936, Alan Turing proved that the halting problem is unsolvable by a computer

That is, the mathematical function *halts* is uncomputable

$$halts : Programs \times \mathbb{N} \rightarrow \{0, 1\},$$
$$halts(P, n) = \begin{cases} 1 & \text{if } P \text{ halts on input } n \\ 0 & \text{otherwise} \end{cases}$$

We proved that *halts* is uncomputable in Python, but our reasoning applies to all languages

It is a fundamental limitation of all computers and programming languages

Uncomputable Functions

It gets worse; not only can we not determine programmatically whether or not a given program halts, we can't determine *anything* "interesting" about the *behavior* of a program in general

For example, suppose we had a program **`prints_something`** that determines whether or not a given program prints something to the screen when run on a specific input:

Then we can write **`halts`**:

```
def halts(fn, i):
    delete all print calls from fn
    replace all returns in fn with prints
    return prints_something(fn, i)
```

Since we know we can't write **`halts`**, our assumption that we can write **`prints_something`** is false

Consequences

There are vast consequences from the impossibility of computing *halts*, or any other sufficiently interesting mathematical functions on programs

The best we can do is approximation

For example, perfect anti-virus software is impossible

- Anti-virus software must either miss some viruses (false negatives), mark some innocent programs as viruses (false positives), or fail to terminate on others

We can't write perfect security analyzers, optimizing compilers, etc.

Incompleteness Theorem

In 1931, Kurt Gödel proved that any mathematical system that contains the theory of non-negative integers must be either *incomplete* or *inconsistent*

- A system is *incomplete* if there are true facts that cannot be proven
- A system is *inconsistent* if there are false claims that can be proven

A proof is just a sequence of statements, which can be represented as bits

- We can generate all proofs the same way we generated all programs

It is also possible to check the validity of a proof using a computer

- Given a finite set of axioms and inference rules, a program can check that each statement in a proof follows from the previous ones

Thus, if a valid proof exists for a mathematical formula, then a computer can find it

Incompleteness Theorem

Given a sufficiently powerful mathematical system, we can write the following formula, which is a predicate form of the *halts* function:

$$H(P, n) = \text{"program } P \text{ halts on input } n\text{"}$$

If *H(P, n)* is provable or disprovable for all *P* and *n*, then we can write a program to prove or disprove it by generating all proofs and checking each one to see if it proves or disproves *H(P, n)*

But then this program would solve the halting problem, which is impossible

Thus, there must be values of *P* and *n* for which *H(P, n)* is neither provable nor disprovable, or for which an incorrect result can be proven

Thus, there are fundamental limitations not only to computation, but to mathematics itself!

Interpretation in Python

`eval`: Evaluates an expression in the current environment and returns the result. Doing so may affect the environment.

`exec`: Executes a statement in the current environment. Doing so may affect the environment.

```
eval('2 + 2')

exec('def square(x): return x * x')
```

`os.system('python <file>')`: Directs the operating system to invoke a new instance of the Python interpreter.