Unification (https://eli.thegreenplace.net/2018/unification/)

November 12, 2018 at 05:49 **Tags** Python (https://eli.thegreenplace.net/tag/python) , Programming (https://eli.thegreenplace.net/tag/programming) , Math (https://eli.thegreenplace.net/tag/math)

In logic and computer science, unification is a process of automatically solving equations between symbolic terms. Unification has several interesting applications, notably in logic programming and type inference (https://eli.thegreenplace.net/2018/type-inference/). In this post I want to present the basic unification algorithm with a complete implementation.

Let's start with some terminology. We'll be using *terms* built from constants, variables and function applications:

- A lowercase letter represents a constant (could be any kind of constant, like an integer or a string)
- An uppercase letter represents a variable
- `f(...)` is an application of function `f` to some parameters, which are *terms* themselves

This representation is borrowed from first-order logic (https://en.wikipedia.org/wiki/First-order_logic) and is also used in the Prolog programming language. Some examples:

- `V`: a single variable term
- `foo(V, k)`: function `foo` applied to variable V and constant k
- `foo(bar(k), baz(V))`: a nested function application

Pattern matching

Unification can be seen as a generalization of *pattern matching*, so let's start with that first.

We're given a constant term and a pattern term. The pattern term has variables. Pattern matching is the problem of finding a variable assignment that will make the two terms match. For example:

- Constant term: `f(a, b, bar(t))`
- Pattern term: `f(a, V, X)`

Trivially, the assignment `V=b` and `X=bar(t)` works here. Another name to call such an assignment is a *substitution*, which maps variables to their assigned values. In a less trivial case, variables can appear multiple times in a pattern:

- Constant term: `f(top(a), a, g(top(a)), t)`
- Pattern term: `f(V, a, g(V), t)`

Here the right substitution is `V=top(a)`.

Sometimes, no valid substitutions exist. If we change the constant term in the latest example to `f(top(b), a, g(top(a)), t)`, then there is no valid substitution becase V would have to match `top(b)` and `top(a)` simultaneously, which is not possible.

Unification

Unification is just like pattern matching, except that both terms can contain variables. So we can no longer say one is the pattern term and the other the constant term. For example:

- First term: `f(a, V, bar(D))`
- Second term `f(D, k, bar(a))`

Given two such terms, finding a variable substitution that will make them equivalent is called *unification*. In this case the substitution is `{D=a, V=k}`.

Note that there is an infinite number of possible unifiers for some solvable unification problem. For example, given:

- First term: `f(X, Y)`
- Second term: `f(Z, g(X))`

We have the substitution `{X=Z, Y=g(X)}` but also something like `{X=K, Z=K, Y=g(K)}` and `{X=j(K), Z=j(K), Y=g(j(K))}` and so on. The first substitution is the simplest one, and also the most general. It's called the *most general unifier* or *mgu*. Intuitively, the *mgu* can be turned into any other unifier by performing another substitution. For example `{X=Z, Y=g(X)}` can be turned into `{X=j(K), Z=j(K), Y=g(j(K))}` by applying the substitution `{Z=j(K)}` to it. Note that the reverse doesn't work, as we can't turn the second into the first by using a substitution. So we say that `{X=Z, Y=g(X)}` is the most general unifier for the two given terms, and it's the *mgu* we want to find.

An algorithm for unification

Solving unification problems may seem simple, but there are a number of subtle corner cases to be aware of. In his 1991 paper Correcting a Widespread Error in Unification Algorithms (https://www.semanticscholar.org/paper/Correcting-a-Widespread-Error-in-Unification-Norvig/95af3dc93c2e69b2c739a9098c3428a49e54e1b6), Peter Norvig noted a common error that exists in many books presenting the algorithm, including SICP.

The correct algorithm is based on J.A. Robinson's 1965 paper "A machine-oriented logic based on the resolution principle". More efficient algorithms have been developed over time since it was first published, but our focus here will be on correctness and simplicity rather than performance.

The following implementation is based on Norvig's, and the full code (with tests) is available on Github (https://github.com/eliben/code-for-blog/blob/master/2018/unif/unifier.py). This implementation uses Python 3, while Norvig's original is in Common Lisp. There's a slight difference in representations too, as Norvig uses the Lisp-y `(f X Y)` syntax to denote an application of function `f`. The two representations are isomorphic, and I'm picking the more classical one which is used in most papers on the subject. In any case, if you're interested in the more Lisp-y version, I have some Clojure code online (https://github.com/eliben/paip-in-clojure/tree/master/src/paip/11_logic) that ports Norvig's implementation more directly.

We'll start by defining the data structure for terms:

```
class Term:
    pass

class App(Term):
    def __init__(self, fname, args=()):
       self.fname = fname
       self.args = args

    # Not shown here: __str__ and __eq__, see full code for the details...


class Var(Term):
    def __init__(self, name):
        self.name = name


class Const(Term):
    def __init__(self, value):
        self.value = value
```

An `App` represents the application of function `fname` to a sequence of arguments.

```
def unify(x, y, subst):
    """Unifies term x and y with initial subst.

    Returns a subst (map of name->term) that unifies x and y, or None if
    they can't be unified. Pass subst={} if no subst are initially
    known. Note that {} means valid (but empty) subst.
    """
    if subst is None:
        return None
    elif x == y:
        return subst
    elif isinstance(x, Var):
        return unify_variable(x, y, subst)
    elif isinstance(y, Var):
        return unify_variable(y, x, subst)
    elif isinstance(x, App) and isinstance(y, App):
        if x.fname != y.fname or len(x.args) != len(y.args):
            return None
        else:
            for i in range(len(x.args)):
                subst = unify(x.args[i], y.args[i], subst)
            return subst
    else:
        return None
```

`unify` is the main function driving the algorithm. It looks for a *substitution*, which is a Python dict mapping variable names to terms. When either side is a variable, it calls `unify_variable` which is shown next. Otherwise, if both sides are function applications, it ensures they apply the same function (otherwise there's no match) and then unifies their arguments one by one, carefully carrying the updated substitution throughout the process.

```
def unify_variable(v, x, subst):
    """Unifies variable v with term x, using subst.

    Returns updated subst or None on failure.
    """
    assert isinstance(v, Var)
    if v.name in subst:
        return unify(subst[v.name], x, subst)
    elif isinstance(x, Var) and x.name in subst:
        return unify(v, subst[x.name], subst)
    elif occurs_check(v, x, subst):
        return None
    else:
        # v is not yet in subst and can't simplify x. Extend subst.
        return {**subst, v.name: x}
```

The key idea here is recursive unification. If `v` is bound in the substitution, we try to unify its definition with `x` to guarantee consistency throughout the unification process (and vice versa when `x` is a variable). There's another function being used here - `occurs_check`; I'm retaining its classical name from early presentations of unification. Its goal is to guarantee that we don't have self-referential variable bindings like `X=f(X)` that would lead to potentially infinite unifiers.

```
def occurs_check(v, term, subst):
    """Does the variable v occur anywhere inside term?

    Variables in term are looked up in subst and the check is applied
    recursively.
    """
    assert isinstance(v, Var)
    if v == term:
        return True
    elif isinstance(term, Var) and term.name in subst:
        return occurs_check(v, subst[term.name], subst)
    elif isinstance(term, App):
        return any(occurs_check(v, arg, subst) for arg in term.args)
    else:
        return False
```

Let's see how this code handles some of the unification examples discussed earlier in the post. Starting with the pattern matching example, where variables are just one one side:

```
>>> unify(parse_term('f(a, b, bar(t))'), parse_term('f(a, V, X)'), {})
{'V': b, 'X': bar(t)}
```

Now the examples from the *Unification* section:

```
>>> unify(parse_term('f(a, V, bar(D))'), parse_term('f(D, k, bar(a))'), {})
{'D': a, 'V': k}
>>> unify(parse_term('f(X, Y)'), parse_term('f(Z, g(X))'), {})
{'X': Z, 'Y': g(X)}
```

Finally, let's try one where unification will fail due to two conflicting definitions of variable X.

```
>>> unify(parse_term('f(X, Y, X)'), parse_term('f(r, g(X), p)'), {})
None
```

Lastly, it's instructive to trace through the execution of the algorithm for a non-trivial unification to see how it works. Let's unify the terms `f(X,h(X),Y,g(Y))` and `f(g(Z),W,Z,X)`:

- `unify` is called, sees the root is an `App` of function `f` and loops over the arguments.
    - `unify(X, g(Z))` invokes `unify_variable` because `X` is a variable, and the result is augmenting subst with `X=g(Z)`
    - `unify(h(X), W)` invokes `unify_variable` because `W` is a variable, so the subst grows to `{X=g(Z), W=h(X)}`
    - `unify(Y, Z)` invokes `unify_variable`; since neither `Y` nor `Z` are in subst yet, the subst grows to `{X=g(Z), W=h(X), Y=Z}` (note that the binding between two variables is arbitrary; `Z=Y` would be equivalent)
    - `unify(g(Y), X)` invokes `unify_variable`; here things get more interesting, because `X` is already in the subst, so now we call `unify` on `g(Y)` and `g(Z)` (what `X` is bound to)
        - The functions match for both terms (`g`), so there's another loop over arguments, this time only for unifying `Y` and `Z`
        - `unify_variable` for `Y` and `Z` leads to lookup of `Y` in the subst and then `unify(Z, Z)`, which returns the unmodified subst; the result is that nothing new is added to the subst, but the unification of `g(Y)` and `g(Z)` succeeds, because it agrees with the existing bindings in subst
- The final result is `{X=g(Z), W=h(X), Y=Z}`

Efficiency

The algorithm presented here is not particularly efficient, and when dealing with large unification problems it's wise to consider more advanced options. It does too much copying around of subst, and also too much work is repeated because we don't try to cache terms that have already been unified.

For a good overview of the efficiency of unification algorithms, I recommend checking out two papers:

- "An Efficient Unificaiton algorithm" by Martelli and Montanari
- "Unification: A Multidisciplinary survey" by Kevin Knight

For comments, please send me an email (mailto:eliben@gmail.com).

Back to top