28 Unification Algorithm

Drew McDermott drew.mcdermott@yale.edu

2015-11-20, 11-30, 2016-10-28 — Yale CS470/570

1 Preliminaries

In the previous note we defined subsumption in terms of *subsumption homomorphism,* a mapping from one F-dag to another that preserves its structure. The same idea is used in our algorithm for finding the *most general subsumee* of two F-dags G_1 and G_2, that is, a G such that $G_1 \sqsubseteq G$ and $G_2 \sqsubseteq G$, and there is no G' such that $G_1 \sqsubseteq G'$ and $G_2 \sqsubseteq G'$ and $G'G \sqsubseteq$. We use the terms *unifier* for the most general subsumee and *unification* for the process required to find it.

The unification algorithm will compute G by building a subsumption homomorphism u from the nodes of the two DAGs G_1 and G_2 to the nodes of a new graph. We extend the notation to define the result of applying a subsumption homomorphism to a graph. Let $u(G)$ be the graph whose nodes are the set $\{D : \exists D' \in nodes(G_1) \cup nodes(G_2) s.t. f(D') = D\}$; and which has an edge from D_1 to D_2 if and only if $D_1 = u(D'_1)$ and $D_2 = u(D'_2)$ and there is an edge from D'_1 to D'_2 in G. When the algorithm is done, the subsumption graph $u(G_1) = u(G_2)$ is the unification of G_1 and G_2.

We can represent the current version of u by a Scala `UnifyMap` data structure. As information about u accumulates, the algorithm creates a succession of `UnifyMap`s, each containing the latest version.

Whenever the algorithm encounters a node f, it checks to see if it already has an image in the current `UnifyMap` u. If so, it uses the image $f' = u(f)$ rather than f. Its current efforts may result in a new `UnifyMap` u', in which f' has an image $f'' = u'(f')$. In addition, $u'(f)$ must $= f''$ as well, because f'' reflects all the information the algorithm has accumulated about f.

In the actual code, we write $u(f)$ as u`.get(`f`)`:

```
class UnifyMap(val map: HashMap[Fstructure, Fstructure] =
                 new HashMap()) {
  def get1(saf: Fstructure): Option[Fstructure] =
    map.get(saf)

  /** The normal thing we want to retrieve is the last
    * non-None item in a chain get1(get1(...get1(saf)))
    */
  def get(saf: Fstructure): Fstructure = {
```

```
      get1(saf) match {
        Some(safx) => get(safx)
        None => saf
      }
    }
  }
```

When we talk about the equality relation on nodes of a graph, what exactly are we talking about? This is always a crucial question about a datatype, although usually the answer is straightforward. In Scala, case classes have a simple answer to the question: Two instances of a case class are equal just in case their pieces are equal. If `C` is such a class, then $\mathtt{C}(a_{11}, \ldots, a_{1k}) = \mathtt{C}(a_{21}, \ldots, a_{2k})$ if and only if $a_{1i} = a_{2i}$, $1 \leq i \leq k$.

But that is precisely what we don't want to happen for subgraphs of an F-DAG. To quote Shieber (p. 10), "We must carefully distinguish between two features with one value and the weaker notion of two features with two different but *similar* values." What makes two values "different" as opposed to "one"? You can point to blobs of ink on the page and appeal to the idea of "same blobs" vs. "blobs with similar feature values," but how does that translate into something inside the computer?

The obvious translation is to interpret "one value," "one token [of a type]" as "one machine address." Scala allows you to do that, by using the function `(_.eq(_))` to compare tokens. `a.eq(b)` if and only if `a` and `b` "are" stored at the same place in memory. There's no way to *define* this relation; you just have to be "in the know," as it were.[1]

Because `eq` seems so out of place in a functional setting, we look for an alternative way of defining sameness of DAG nodes. The simplest way is to give every node a *serial number,* an `Int` that uniquely identifies it. A node in an F-DAG is then represented a wrapped datum of type `PreFstructure`, the wrapping including a `SerialNum`:

```
class Fstructure private(val serial: SerialNum,
                         val actual: PreFstructure) { ...
}
```

The `private` modifier applies to the constructor for `Fstructure`s. We can't allow just anyone to make up a serial number and create an `Fstructure`

[1] You also have to have faith that at runtime it never happens that a copy of an object is made for some obscure efficiency reason. Because the copy doesn't officially exist, if anyone asks if it's the "same" as the original, the answer should be Yes. Should `eq` cover this case?

with that number. The class `FScounter` contains the only legal call to the constructor, and it always increments a global counter to generate the next value it uses.

So now we can give a tidy answer to the question about `Fstructures`: $f_1 = f_2$ if and only if f_1`.serial` $=$ f_2`.serial`.

Unfortunately, that is the last tidy thing about the global counter. In order to avoid side effects, every function that might create a new `Fstructure` must take the global counter as an argument and return an updated counter as part of the value. Furthermore, we have to make sure that once a function has updated counter it never tries to use the original one. We can certainly arrange to throw an exception if a counter is incremented more than once, but it would be nice to have a way of making sure at compile time that the counters are properly handed around.[2]

The serial numbers are not the same as the boxed numbers in linguists' attribute-value matrices (the kind we print as `=`i), which are called "indices." However, the line between the two is a bit blurry, because linguists sometimes use the word "index" so it seems a bit closer to semipermanent serial numbers than to transient boxed numbers. An "index" in the linguists' sense is an abstract entity between syntax and semantics that specifies which entities equal which other entities. For example, consider these two sentences:

1. *John$_i$ shaved him$_i$.

2. John$_i$ shaved him$_j$.

Actually, these are the *same* sentence, so how can it receive an asterisk (meaning "ungrammatical," "syntactically illegal") in version 1, but no asterisk in version 2? Because of the pattern of subscripts. Version 1 gives "John" and "him" the same index; version 2 gives them different indices. Compare these two:

1. The imams$_i$ saw the nurses$_j$ shave each other$_j$.

2. *The imams$_i$ saw the nurses$_j$ shave each other$_i$.

Although both readings are implausible, at least imams have plenty of hair in plain view, and they each *might* have seen each other get shaved by the same team of nurses. Even so, the rules of the English reciprocal pronoun "each other" forbid the second reading completely, at least in my dialect, and I suspect I'm not unusual. Linguists use the term *the binding theory*

[2]There *is* a way of ensuring this: use a monad to manage the state. Unfortunately, this would require reorganizing the code substantially and making it (even) less readable.

to refer to the rules that determine which patterns of coindexation are legal (not that there is any consensus about those rules.)

In practical natural-language processing, a pressing issue is deciding when two noun phrases refer to the same entity. For example, a newspaper article about a group of Muslim clergymen touring an (electric razor?) factory might be of interest to the company owning the factory, assuming they want to track how their products are mentioned in the press. If the first sentence gives the name of the group as "Muslim Clergy Association of Greater Tehran," and a later sentence refers to "the imams," it's important to decide that "Association$_i$" and "imams$_i$" are the same entity — their internal representations have the same serial number (here schematized as i). Although later sentences might cause the program to reverse that decision, the assimilation of those sentences will be seriously hampered unless such decisions are usually made correctly.

Let's move on to a different topic: how unification is to cope with structure sharing. I casually said when introducing `UnifyMap`s that the domain of a `UnifyMap` was $nodes(G_1) \cup nodes(G_2)$, where G_1 and G_2 are the two DAGs being unified. If G_1 and G_2 share nodes, then we're going to run into trouble. The same node can occur in both graphs, but play different roles. For instance, it might occur at depth 1 in one graph and depth 2 in the other, but the image in the `UnifyMap` of a node f must have the same depth in the result graph as f had originally, which results in absurdity. I emphasize that this is just one way of dramatizing the problem. Bugs will pop up all over the place if we don't confront it.

In order to present an example, I have to explain how the "attribute-valued matrices" (AVMs) are actually printed and perhaps read. The vertically oriented notation is awkward, so we will "horizontalize" it. We'll also switch to curly braces to emphasize that the order of attributes is unimportant. Instead of boxes, we'll have a number prefixed by an equal sign ("`=`"). If the number labels an actual AVM, we'll write another equal sign after the number. So the notation

```
{A: Adj, ..., B: {..., C: =1, ...},
         ..., D: =1= {...}, ...}}
```

describes an `Fstructure` with, among other features, an `A` with value `Adj`, a `B`, and a `D`. The value of `B` is an `Fstructure` whose `C` feature's value is the same as the `D` of the whole thing. If r is the top node, then $r(\langle \mathtt{D} \rangle) = r(\langle \mathtt{B,\ C} \rangle)$. We will usually drop the parens when the path is explicit, and just write $r\langle \mathtt{D} \rangle = r\langle \mathtt{B,\ C} \rangle$.[3]

[3]Shieber puts the node inside the angle brackets, and omits commas and so would write

Now consider this `Fstructure`:

$S =$`{A: =1,`
`B: {A: C, B: =1={?}}}`

and consider two F-DAGs, G_1 consisting of (rooted at) S, and G_2 rooted at $S\langle$`B`$\rangle$. (`C` is some leaf constant.) In other words, the structure of G_2 occurs as a piece of G_1. So G_2's root, with depth 0 in G_2, has depth 1 in G_1, thus raising the issue mentioned earlier.

Do they unify and what is the result? If we create a new F-DAG that is isomorphic to G_2, G_2', rooted at

$S' =$ {A: C, B: {?}}

then clearly G_1 and G_2' unify, with result:

$S =$`{A: =1,`
`B: {A: C, B: =1= C}}`

But suppose we do not copy S(`<B>`) before unifying. All unification algorithms work by systemtically examining pieces of the two trees and imposing the constraints that subpieces be equal. (They differ in the order in which they examine the pieces, which can affect asymptotic complexity.) At some point the constraint that $G_1\langle$`B`$\rangle = G_2\langle$`B`$\rangle$ will be considered. But, because $G_2 = G_1$(`<B>`), this will be the constraint that $G_1\langle$`B`$\rangle) = G_1\langle$`B`$\rangle\langle$`B`$\rangle = G_1\langle$`B, B`$\rangle$, which can't be satisfied without making the graph cyclic. Hence the unification should fail.

It's clearly absurd for an accident of structure sharing to determine whether a unification succeeds or fails. There are two ways to deal with this problem:

1. Forbid the two arguments to the unification algorithm from overlapping. In practice, the only way to guarantee that is to copy one of the arguments before the algorithm begins.

2. Label each argument with a 1-bit tag (distinguishing "left" from "right") and keep track of these tags as you go. So the algorithm could distinguish ("left", n) from ("right", n), node n considered as being from the "left" argument from node n considered as being from the "right" argument.

$\langle r$`D`$\rangle = \langle r$`B C`$\rangle$, a notation with little to recommend it except for its resemblance to Lisp.

I don't know which of these two ideas has been used most often, or which is most efficient or leads to the most debuggable code. It's conceivable that a "lazy" copying strategy could allow the system to use idea 1, but not pay too heavy a price for copying a large structure when the unification fails quickly.

We will adopt the simplest approach and copy the right argument before attempting unification.

2 The Algorithm

The main entry point in the actual code is `dagUnify`, which copies its right argument (it could have been the left), and calls `unify`. This function creates a `UnifyMap` and calls `unifyMapPursue` to complete it.

```
private def unifyMapPursue(fsx: Fstructure, fsy: Fstructure, um: UnifyMap):
    Option[UnifyMap] = {

  val fsxI = um.get(fsx)
  val fsyI = um.get(fsy)

  (fsxI.actual, fsyI.actual) match {
    case (FSVar, \_) => unifyMapCycleCheck(um, fsxI, fsyI)
    case (\_, FSVar) => unifyMapCycleCheck(um, fsyI, fsxI)
    case (l1: FSleaf, l2: FSleaf) => {
      if (l1.v == l2.v) unifyMapNodesMerge(um, fsxI, fsyI, fsyI)
      else None
    }
    case (c1: FScomplex, c2: FScomplex) => {
      if (c1.sort == c2.sort) {
        for {
          fs <- c1.sort.featSorts
          (finalFeatMap, updatedUnifyMap) <-
            featuresPursue(fs.map(\_.\_1), c1, c2, um)
          c = new FScomplex(c2.sort, finalFeatMap)
        } yield {
          val (newCtr, newFstructure) =
            updatedUnifyMap.ctr.stampFstructure(
              new FScomplex(c2.sort, finalFeatMap)
            )
          unifyMapNodesMerge(
```

```
            new unifyMap(newCtr, updatedUnifyMap.map),
            fsxI, fxyI, newFstructure
          )
        }
      }
      else None // -- Let's say
    }
  // Leaves don't matches complexes --
  case \_ => None
}}
```

This function runs some checks to see if the nodes `fsx` and `fsy` are unifiable, possibly constructing a new F-structure, then calls `unifyMapNodesMerge` to augment `um`, the `unifyMap`, with link from `fsx` and `fsy` to the new F-structure. So from now on, if `fsx` or `fsy` iss encountered again, the algorithm will treat them as if they were this new node.

For this description to be accurate, `unifyMapPursue` has to work with `fsxI` and `fsyI`, the *images* of `fsx` and `fsy` at the outset. Plus, there is one inaccuracy to be fixed. The algorithm constructs a new node when it unifies two complex F-structures, but when comparing leaves, including variables, it doesn't generate a new node, but just causes one of the old nodes to point to the other in the revised `UnifyMap`.

The checks to make sure the two nodes are unifiable boil down to these:

- If both `fsxI` and `fsyI` are leaves, they must have the same value. If they pass this test, the current `UnifyMap`, `um`, can be returned unchanged. `I`f one or both of `fsxI` and `fsyI` is a variable, then one of the variables is mapped to the other node in the augmented version of `um` that is returned. But first the algorithm must check whether adding this link would introduce a cycle into the F-DAG. If so, the entire unification attempt fails. `I`f both of `fsxI` and `fsyI` are "complex," then each has a *sort* and a *feature map*. The sort determines what features are legal at this kind of node. The two nodes match if they have the same sort and if every feature licensed by the sort has a value in one node that is unifiable with the value in th eother. This latter check is implemented in `featuresPursue`.

The most expensive check is the search for a potential cycle when a variable is matched with a non-variable. In an F-structure such as this one:

```
{A: =1,B: =1= {A: =2, B: =2= {A: =3 ...
```

{A: =n, B: =n= {C: 42}}}}}

there are 2^n paths from the F-structure down to the substructure with feature C.