title: Unification-fd tutorial (part 1/n) tags: haskell, haskell planet, unification

A while back I released the *unification-fd* library, which gives a generic implementation of first-order unification of non-cyclic terms. I've given a few talks on how the library is implemented and what optimizations it performs, but that's not the topic for today. Today, I'm going to talk about how to *use* the library.

Unification is a widely useful operation and, consequently, comes in many different flavors. The version currently supported by the library is the sort used by logic programming languages like Prolog, Curry, Dyna, and MiniKanren; which is the same sort that's used for unification-based type inference algorithms like Hindleyâ€“Damasâ€“Milner. Of these two examples, the logic programming example is the simpler one to discussâ€” at least for folks who've used a language like Prolog before. So let's start from there.

*Caveat Emptor:* This post is something of a stream of consciousness. I've gotten a few requests for tutorials on how to use the library, hence the post. But the requests weren't terribly specific about what problems they've had or what's been difficult to figure out, so I'm shooting in the dark as far as what folks need and how much background they have. I'm going to assume you're familiar with Prolog and the basics of what unification is and does.

**Logic Terms**

A ***term***, in Prolog, is just a fancy name for a value of some algebraic data type. In most variants of Prolog there's no explicit definition of the ADT, no restriction on what the constructors are, and no type checking to ensure that subterms have a particular shape. That is, Prolog is what's called a single-sorted logic; in other words Prolog is an untyped or unityped language. With *unification-fd* we can implement multi-sorted (aka typed) logics, but for this tutorial we're going to stick with Prolog's single-sorted approach.

Opening up Control.Unification we'll see a handful of types and type classes, followed by a bunch of operators. The UTerm data type captures the recursive structure of logic terms. (UTerm is the free monad, if you're familiar with that terminology.) That is, given some functor t which describes the constructors of our logic terms, and some type v which describes our logic variables, the type UTerm t v is the type of logic terms: trees with multiple layers of t structure and leaves of type v. For our single-sorted logic, here's an implementation of t:

data T a = T String [a]

The String gives the name of the term constructor, and the list gives the ordered sequence of subterms. Thus, the Prolog term foo(bar,baz(X)) would be implemented as UTerm$T "foo" [UTerm$T "bar" [], UTerm$T "baz" [UVar x]]. If we're going to be building these terms directly, then we probably want to define some smart constructors to reduce the syntactic noise:

foo x y = UTerm$T "foo" [x,y]

bar = UTerm$T "bar" []

baz x = UTerm$T "baz" [x]

Now, we can implement the Prolog term as foo bar (baz x). If you prefer a more Prolog-like syntax, you can use uncurried definitions for these smart constructors.

**Unifiable**

In order to use our T data type with the rest of the API, we'll need to give a Unifiable instance for it. Before we do that we'll have to give Functor, Foldable, and Traversable instances. These are straightforward and can be automatically derived with the proper language pragmas.

The Unifiable class gives one step of the unification process. Just as we only need to specify one level of the ADT (i.e., T) and then we can use the library's UTerm to generate the recursive ADT, so too we only need to specify one level of the unification (i.e., zipMatch) and then we can use the library's operators to perform the recursive unification, subsumption, etc.

The zipMatch function takes two arguments of type t a. The abstract t will be our concrete T type. The abstract a is polymorphic, which ensures that we can't mess around with more than one level of the term at once. You can think of it as if a is UTerm T v, thus the two arguments have the type T (UTerm T v). In other words, they're values of type UTerm T v with the guarantee that neither one is a variable.

The zipMatch method has the rather complicated return type: Maybe (t (Either a (a,a))). Let's unpack this a bit by thinking about how unification works. When we try to unify two terms, first we look at their head constructors. If the constructors are different, then the terms aren't unifiable, so we return Nothing to indicate that unification has failed. Otherwise, the constructors match and so we have to recursively unify the subterms. Since the T structures of the two terms match, we can return Just t0 where t0 has the same T structure as both input terms. In order to indicate that we still have to recursively unify subterms, we fill t0 with Right(l,r) values where l is a subterm of the first argument to zipMatch and r is the corresponding subterm of the second argument. Thus, zipMatch is a generalized zipping function for combining the shared structure and pairing up the substructures. And now, the implementation:

instance Unifiable T where

zipMatch (T m ls) (T n rs)

| m /= n = Nothing

| otherwise =

T n <$> pairWith (\l r -> Right(l,r)) ls rs

Where [list-extras:Data.List.Extras.Pair.pairWith](http://hackage.haskell.org/package/list-extras-0.4.1.3/docs/Data-List-Extras-Pair.html) is a version of zip which returns Nothing if the lists have different lengths. So, if the names m and n match, and if the two arguments have the same number of subterms, then we pair those subterms off in order; otherwise, the names or the lengths don't match, so we return Nothing.

**Feature Structures**

For the T example, we don't need to worry about the Left option. The reason it's there is to support feature structures and other sparse representations of terms. That is, consider the following type:

newtype FS k a = FS (Map k a)

Using this type, our logic terms are sets of keyâ€“subterm pairs. When unifying maps like these, what do we do if one argument has a binding for a particular key but the other argument does not? In the T example we assumed that subterms which couldn't be paired together (because the lists were different lengths) meant the unification must fail. But for FS it makes more sense to assume that terms which can't be paired up automatically succeed! That is, we'd like to assume that all the keys which are not explicitly present in the Map k a are implicitly present and bound to some unique logic variable. Since the unique logic variables are implicit, there's no need to actually keep track of them, we'll just implicitly unify them with the subterm that can't be paired off.

This may make more sense if you see the Unifiable instance:

instance (Ord k) => Unifiable (FS k) where

zipMatch (FS ls) (FS rs) =

Just . FS $ unionWith (\(Left l) (Left r) -> Right(l,r))

(fmap Left ls)

(fmap Left rs)

We start off by mapping Left over both the ls and the rs. We then call unionWith to pair things up. For any given key, if both ls and rs specify a subterm, then these subterms will be paired up as Right(l,r). If, however, we have left over subterms from either ls or rs, then we keep them around as Left l or Left r. Thus, the Unifiable instance for FS performs a union of the FS structure, whereas the instance for T performs an intersection of T structure.

The Left option can be used in any situation where you can immediately resolve the unification of subterms, whereas the Right option says you still have work to do.

**Logic Variables**

The library ships with two implementations of logic variables. The IntVar implementation uses Int as the names of variables, and uses an IntMap to keep track of the environment. The STVar implementation uses STRefs, so we can use actual mutation for binding logic variables, rather than keeping an explicit environment around. Of course, mutation complicates things, so the two implementations have different pros and cons.

Performing unification has the side effect of binding logic variables to terms. Thus, we'll want to use a monad in order to keep track of these effects. The BindingMonad type class provides the definition of what we need from our ambient monad. In particular, we need to be able to generate fresh logic variables, to bind logic variables, and to lookup what our logic variables are bound to. The library provides the necessary instances for both IntVar and STVar.

You can, of course, provide your own implementations of Variable and BindingMonad. However, doing so is beyond the scope of the current tutorial. For simplicity, we'll use the IntVar implementation below.

**Example Programs**

When embedding Prolog programs into Haskell, the main operators we want to consider are those in the section titled "Operations on two terms". These are structural equality (i.e., equality modulo substitution), structural equivalence (i.e., structural equality modulo alpha-variance), unification, and subsumption.

Consider the following Horn clause in Prolog:

example1(X,Y,Z) :- X = Y, Y = Z.

To implement this in Haskell we want a function which takes in three arguments, unifies the first two, and then unifies the second two. Thus,[1](http://community.haskell.org/%7Ewren/unification-fd/test/tutorial/tutorial1.html" \l "XXXXX:fn1)

example1 x y z = do

x =:= y

y =:= z

To run this program we'd use one of the functions runIntBindingT, evalIntBindingT, or execIntBindingT, depending on whether we care about the binding state, the resulting logic term, or both. Of course, since the unifications may fail, we also need to run the underlying error monad using something like runErrorT[2](http://community.haskell.org/%7Ewren/unification-fd/test/tutorial/tutorial1.html" \l "XXXXX:fn2)[3](http://community.haskell.org/%7Ewren/unification-fd/test/tutorial/tutorial1.html" \l "XXXXX:fn3). And since these are both monad transformers, we'll need to use runIdentity or the like in order to run the base monad. Thus, the functions to execute the entire monad stack will look like:

-- Aliases to simplify our type signatures. N.B., the signatures are not actually required to get things to typecheck.

type PrologTerm = UTerm T IntVar

type PrologFailure = UnificationFailure T IntVar

type PrologBindingMonad = IntBindingT T Identity

type PrologBindingState = IntBindingState T

type PrologFallibleMonad = ErrorT PrologFailure PrologBindingMonad

runProlog

:: PrologFallibleMonad a

-> (Either PrologFailure a, PrologBindingState)

runProlog = runIdentity . runIntBindingT . runErrorT

evalProlog

:: PrologFallibleMonad a

-> Either PrologFailure a

evalProlog = runIdentity . evalIntBindingT . runErrorT

execProlog

:: PrologFallibleMonad a

-> PrologBindingState

execProlog = runIdentity . execIntBindingT . runErrorT

Here are some more examples:

-- A helper function to reduce boilerplate. First we get a free variable, then we embed it into @PrologTerm@, and then we embed it into some error monad (for capturing unification failures).

getFreeVar = lift (UVar <$> freeVar)

-- example2(X,Z) :- X = Y, Y = Z.

example2 x z = do

y <- getFreeVar

x =:= y

y =:= z

-- example3(X,Z) :- example1(X,Y,Z).

example3 x z = do

y <- getFreeVar

example1 x y z

-- example4(X) :- X = bar; X = backtrack.

example4 x = (x =:= bar) <|> (x =:= atom "backtrack")

The complete code for this post can be found [here](http://community.haskell.org/%7Ewren/unification-fd/test/tutorial/tutorial1.html). That file includes some helpful type aliases to help clarify a bit what's going on, as well as the functions for executing the Prolog monad used in these examples.

**Term Factoring and Clause Resolution Automata (CRAs)**

Note that for the above examples, the Haskell functions only execute the right-hand side of the Horn clause. In Prolog itself, there's also a process of searching through all the Horn clauses in a program and deciding which one to execute next. A naive way to implement that search process would be to have a list of all the Horn clauses and walk through it, trying to unify the goal with the left-hand side of each clause and executing the right-hand side if it matches. A more efficient way would be to compile all the right-hand sides into a single automaton, allowing us to match the goal against all the right-hand sides at once. (The idea here is similar to compiling a bunch of strings together into a trie or regex.)

Constructing optimal CRAs is NP-complete in general, though it's feasible if we have an arbitrary ordering of clauses like Prolog's topâ€“down order for trying each clause. The *unification-fd* library does not implement any support for CRAs at present, though it's something I'd like to add in the future. For more information on this topic, see [Dawson et al. (1995) *Optimizing Clause Resolution: Beyond Unification Factoring*](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.6932) and [Dawson et al. (1996) *Principles and Practice of Unification Factoring*](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.2895).

**Other operators**

In addition to unification itself, it's often helpful to have various other operators on hand.

One such operator is the subsumption operator. Whereas unification looks for a most-general substitution which when applied to both arguments yields terms which are structurally equal (i.e., l =:= r computes the most general s such that s l === s r), subsumption only applies the substitution to one side. That is, l subsumes r just in case there is a substitution s such that s l === r. In other words, l is more general or less defined than r; or in other words, r is a substitution instance of l. Subsumption shows up in cases where we have to hold r fixed for some reason, such as when implementing polymorphism or subtyping.

Other operators work on just one term, such as determining the free variables in a term, explicitly applying the substitution in order to obtain a pure term, or cloning a term to make a copy where all the free variables have been renamed to fresh variables. These sorts of operators aren't used very often in logic programming itself, but are crucial for implementing logic programming languages.

**Conclusion**

Hopefully that gives a quick idea of how the library's API is set up. Next time I'll walk through an implementation of Hindleyâ€“Damasâ€“Milner type inference, and then higher-ranked polymorphism Ã  la [Peyton Jones et al. (2011) *Practical type inference for arbitrary-rank types*](http://research.microsoft.com/en-us/um/people/simonpj/papers/higher-rank/putting.pdf). If there's still interest after that, I can get into some of the guts of the library's implementationâ€” like ranked path compression, maximal structure sharing, and so on.

If you have any particular questions you'd like me to address, drop me a line.

[1] N.B., a more efficient implementation is:

example1' x y z = do

y' <- x =:= y

y' =:= z

The unification operator returns a new term which guarantees maximal structure sharing with both of its arguments. The implementation of unification makes use of observable structure sharing, so by capturing y' and using it in lieu of y, the subsequent unifications can avoid redundant work. [â†©](http://community.haskell.org/%7Ewren/unification-fd/test/tutorial/tutorial1.html#XXXXX:fnref1)

[2] The ErrorT transformer was deprecated by *transformers-0.4.1.0*, though it still works for this tutorial. Unfortunately, the preferred ExceptT does not work since UnificationFailure doesn't have a Monoid instance as of *unification-fd-0.9.0*. The definition of UnificationFailure already contains a hack to get it to work with ErrorT, but future versions of the library will remove that hack and will require users to specify their own monoid for combining errors. The First monoid captures the current behavior, though one may prefer to use other monoids such as a monoid that gives a trace of the full execution path, or witnesses for all the backtracks, etc. [â†©](http://community.haskell.org/%7Ewren/unification-fd/test/tutorial/tutorial1.html#XXXXX:fnref2)

[3] To be honest, I don't entirely recall why I had the error monad explicitly separated out as a monad transformer over the binding monad, rather than allowing these two layers to be combined. Since it's so awkward, I'm sure there was some reason behind it, I just failed to make note of why. If there turns out to be no decent reason for it, future versions of the library may remove this fine-grain distinction. [â†©](http://community.haskell.org/%7Ewren/unification-fd/test/tutorial/tutorial1.html#XXXXX:fnref3)

[WordPress](http://wordpress.com/press-this.php?u=http%3A%2F%2Fwinterkoninkje.dreamwidth.org%2FXXXXX.html&t=Unification-fd%20tutorial%20%28part%201%2Fn%29&s=)