Variables in Types

15-312, Spring 2017

February 18, 2017

Abstract

We've seen a number of examples in class of types which use variables. Having variables in our type systems lends a great deal of power to languages using these type systems. We're going to look at how variables are used in generic programming, inductive & coinductive types, and polymorphic types.

Variables in Expressions

To explain how variables in types work, let's look more carefully at how we've used variables until now:

```
wild(x.d)              (Card.Deck) Deck
let(e1; x.e2)          (Exp, Exp.Exp) Exp
rec(e; e0; x.y.e1)     (Exp, Exp, Exp.Exp.Exp) Exp
case(e; x1.e1; x2.e2)  (Exp, Exp.Exp, Exp.Exp) Exp
lam(τ; x.e)            (Type, Exp.Exp) Exp
```

Even though each of these syntactic forms uses variables in some way, they all behave differently. Recall at the beginning of the course we asserted:

> Variables are given meaning by substitution.

Each of these operators has a different meaning because of the rules defining it's statics and dynamics. In particular, the rules for each of these operators has **involved a substitution**:

```
let(e1; x.e2)        → [e1 / x]e2
app(lam(τ; x.e); e') → [e' / x]e
...
```

So we can see from the rules what meaning the variables has.

Polynomial Type Operators

For System T with products and sums, the grammar of our *type system* looked like this:

```
τ ::=
  nat()          nat        () Type
  arr(τ1; τ2)    τ1 → τ2    (Type, Type) Type
  unit()         1          () Type
  prod(τ1; τ2)   τ1 × τ2    (Type, Type) Type
  void()         0          () Type
  sum(τ1; τ2)    τ1 + τ2    (Type, Type) Type
```

So we can see that our operators for constructing types have all had valence 0.

Our first foray into adding variables to our types is what we will call "polynomial type operators." Polynomial type operators have the exact same syntax as our types from System T, augmented with variables and binding sites (but for technical reasons, without functions):

```
τ ::=
  t              t
  t.τ            t.τ
  nat()          nat        () Type
  unit()         1          () Type
  prod(τ1; τ2)   τ1 × τ2    (Type, Type) Type
  void()         0          () Type
  sum(τ1; τ2)    τ1 + τ2    (Type, Type) Type
```

An important catch comes along now that we have variables in our program: variables can be either free or bound, so we have to define what a valid polynomial type operator looks like. The complete set of rules can be found in the book, but the important rule looks like this:

$$\frac{}{\texttt{t.t poly}}$$

So if our polynomial type operator has a variable in the body, it must be the variable we declared out front. Put another way, we can only ever use one variable. This makes sense by analogy with polynomials from algebra class: $f(x) = x^2 + 2x + 1$ is valid, but $f(x) = x + z$ is not, because the $f(x)$ part says that we can only use x in our polynomial.

As a side note, I'm not trying to say that we **can't** have polynomial types with more than one variable. We've seen functions like $f(x, y)$ in school before. But for our purposes, if we say "polynomial type operator," there's only one free variable in the τ of $t.\tau$.

Generic Programming

On their own, polynomial type operators aren't all that cool. We have variables, but we haven't given them meaning with substitution!

To this end, one thing we can do with polynomial type operators is use them for *generic programming*. "Generic programming" is just a fancy way of saying that we don't want to have to repeat ourselves.

In particular, one example of generic programming is the generic map function `map{t.τ}(f)(e)`. Intuitively, `map` is just a way of applying a function to all the "interesting" parts of a datatype, where the "interesting" parts are demarcated by free occurrences of t in τ.

The type of map looks like this:

```
map{t.τ} : (ρ → ρ') → [ρ/t]τ → [ρ'/t]τ
```

So what we see going on here is that if we have a function with type $\rho \to \rho'$, then map will take in something which has ρ in all the t placeholders, and return something which has ρ' in all the t placeholders. Another cool thing here is that this substitution happens in the statics! All the examples of variable substitution we had before were in the dynamics. This makes sense; types are a static thing, whereas values are a runtime thing.

The cool part about `map` is that it will work for any polynomial type operator that we write down! There are a lot of types we can write down with just products, sums, and variables. However, there are a bunch that we can't… I'm thinking of types like `list`, `nat`, `tree`, `stream`, etc. These types are recursive, and are not expressible with just products and sums.

Inductive and Coinductive Types

Polynomial types are great, but there are still more kinds of data we'd like to be able to represent in our type system. We can revisit the syntax for polynomial types, and update it to include inductive and coinductive types:

```
τ ::=
  t              t

  unit()         1          () Type
  prod(τ1; τ2)   τ1 × τ2    (Type, Type) Type
  void()         0          () Type
  sum(τ1; τ2)    τ1 + τ2    (Type, Type) Type
```

```
arr(τ1; τ2)    τ1 → τ2    (Type, Type) Type

ind(t.τ)       μ(t.τ)     (Type.Type) Type
coi(t.τ)       ν(t.τ)     (Type.Type) Type
```

Note how `nat`, an inductive type, is not here anymore. This is because we can define it outright without having to special case it into our type system.

Another thing to note from the arities is that the operators `ind` and `coi` actually *create a Type*. In the same sense that `arr` takes in two Types and creates another Type, `ind` and `coi` take in a Type with free var and wrap it in an abstractor to bind the free variable, thus *creating a Type*.

As an example, our language can now have values of type `μ(t. 1 + t)`, or `ν(t. 1 + (μ(s. 1 + s) × t))`. There are two things to note here:

- These strings of symbols are really annoying to work with as a whole, so we often define names for these types, and refer to them by their names rather than their abstract syntax.
- In order to have values of this type, we need corresponding constructs in our language of `Exp`s that let us introduce and eliminate things of this type.

The type `μ(t. 1 + t)` behaves like the natural numbers, so we call it `nat`. This means that `ν(t. 1 + (μ(s. 1 + s) × t))` becomes `ν(t. 1 + (nat × t))`. This type behaves like streams of natural numbers, so we call it `natstream`.

Now let's look at the constructs of our language that we'll use to introduce and eliminate these types:

```
fold{t.τ}   : [μ(t.τ) / t]τ → μ(t.τ)
rec{t.τ}    : ([ρ/t]τ → ρ) → μ(t.τ) → ρ

unfold{t.τ} : ν(t.τ) → [ν(t.τ) / t]τ
gen{t.τ}    : (ρ → [ρ/t]τ) → ρ → ν(t.τ)
```

This is a lot of substitution, so sometimes we write the same thing with function application instead of substitutions because it looks prettier:

```
fold{t.τ}   = fold{φ}   : φ(μ(φ)) → μ(φ)
rec{t.τ}    = rec{φ}    : (φ(ρ) → ρ) → μ(φ) → ρ

unfold{t.τ} = unfold{φ} : ν(φ) → φ(ν(φ))
gen{t.τ}    = gen{φ}    : (ρ → φ(ρ)) → ρ → ν(φ)
```

So again, we can see how the substitution gives meaning to these types, even though it's not entirely obvious at first what that meaning is.

In recitation this week, we saw the type of `fold{t.1 + (t × t)}` (or in words, the instance of `fold` for data-less trees).

If the general form for `fold` is

```
fold{t.τ}           : [μ(t.τ) / t]τ → μ(t.τ)
```

then the `fold` for trees is

```
fold{t.1 + (t × t)} : [μ(t.1 + (t × t)) / t](1 + (t × t)) → μ(t.1 + (t × t))
                    : [tree / t](1 + (t × t)) → tree
                    : 1 + (tree × tree) → tree
```

It becomes readily apparent where the recursive structure of inductive types comes from. We create a tree by giving `fold` either unit or two trees, and it gives us back a tree. We used substitution and type variables to define the types to be recursive in this way.

As another side note, to see more examples of how the mechanics of these substitutions work, check the recitation notes for week 5. They have an example for `fold`, `rec`, `unfold`, and `gen`.

Polymorphic Types

The most recent example of variables in our types that we saw dealt with System F, whose syntax looks like this:

```
τ ::=
  t              t
  arr(τ1; τ2)    τ1 → τ2     (Type, Type) Type
  all(t.τ)       ∀(t.τ)      (Type.Type) Type

e ::=
  lam(τ; x.e)    λ(x:τ). e   (Type, Exp.Exp) Exp
  ap(e1; e2)     e1 (e2)     (Exp, Exp) Exp
  Lam(t.e)       Λt. e       (Type.Exp) Exp
  App(e; τ)      e [τ]       (Exp, Type) Exp
```

We've only briefly touched on polymorphic types, but the idea is that we can now write down functions that work the same regardless of our choice for the types of some parts of the expression.

Keeping with the theme of meaning through substitution, the interesting case for System F is in *type applications*:

$$\frac{\Delta\,\Gamma \vdash e : \forall(t.\tau') \qquad \Delta \vdash \tau\ \mathsf{type}}{\Delta\,\Gamma \vdash e[\tau] : [\tau/t]\tau'}$$

So what this is saying is that if some expression e is valid for all choices of t, then it's definitely valid for the particular choice of τ, provided that τ is indeed a type.

Putting it All Together, and Then Some

We've seen plenty of examples of types with variables, but as one last bit of "food for thought," we now have *almost* all the tools to define parametric data types:

```
  list = λ(α.μ(t. 1 + (α × t)))
α list = ∀(α. list(α))

  tree = λ(α.μ(t. 1 + (t × α × t)))
α tree = ∀(α. tree(α))

  stream = λ(α.ν(t. 1 + (α × t)))
α stream = ∀(α. stream(α))
```

I say *almost* because I cheated a bit and snuck in this `λ(α.τ)` construct that we've never seen. These are "type-level lambdas" and we don't cover them in this course—everything from here on is just for fun.[1]

You'll notice that there's a difference between `list` and `α list`. The former is a type *constructor* while the latter is a type. When we're writing SML, we can have actual values of type `α list`, like `[]` or the `x` in `fn (x : α list) => ...` for example. But we can't have values of type `list`, because it's actually a type constructor, not a type itself.

If we squint, we see that the `α list` we're familiar with from SML uses both polymorphic types (i.e., `∀(t.τ)`) and inductive types (i.e., `μ(t.τ)`). This should match up with what we expect: polymorphic types basically say "contains anything" and inductive types basically say "repeats itself."

If we squint even harder, what we get are type operators!

```
α.μ(t. 1 + (α × t))
α.μ(t. 1 + (t × α × t))
α.ν(t. 1 + (α × t))
```

[1] We don't cover type lambdas in this class (but you will if you take HOT Compilation!) The gist of it is that type-level lambdas take in types and **construct** new types.

But these type operators aren't polynomial type operators. Polynomial type operators only had products and sums, whereas these have inductive and coinductive types. This means that our `map{t.τ}` won't work on them… yet!

If we can manage to define the cases of map for inductive and coinductive types, then we'll automatically have a map function for any polymorphic and recursive data structure we can imagine! In symbols, we need to define these cases:

```
map{α.μ(t.τ)} → ...
map{α.ν(t.τ)} → ...
```

You'll see how to do this on hw03!