15-424/15-624 Recitation 10
Axioms and Uniform Substitutions

1 Why Uniform Substitutions

Whenever we do a proof in KeYmaera X on the labs, we don't have to worry about whether our proof steps are sound: KeYmaera X will yell at us if we try to do something unsound. That puts an awful lot of pressure on KeYmaera X though, don't you think? If there's a bug in one of KeYmaera X's proof rules, that would cause some serious trouble for us, because we'd apply some proof rule expecting KeYmaera X to catch our mistakes and then BOOM we accidentally proved false.

Thus it's important to design our theorem prover so that correctness (soundness, anyway) is as obvious as possible. Ultimately this also means designing our axioms and proof rules so that their soundness is as obvious as possible, too. The way we do this in KeYmaera X is with a proof rule called *uniform substitution*.

2 What is Uniform Substitution

Before we can decide how to make dL simple, we need to think about why dL could get too complicated in the first place. In practice, most of the complexity comes from the *side conditions* for applying axioms: these are any extra preconditions for soundness we have to check that aren't written down explicitly in the axiom. They're named side conditions because they're usually only written down on the side, if at all — that's not a good way to make sure our proofs are sound! As a canonical example, let's think about the vacuous axiom V:

$$\phi \to [\alpha]\phi$$

If ϕ is true now it's true after we run α . . . that doesn't sound like it's always true! And it's not. But there's a useful case where it will be true: if α doesn't modify any of the variables that show up in ϕ, then it also can't modify the fact that ϕ is true. This is our side condition, which we express mathematically as $\mathrm{FV}(\phi) \cap \mathrm{BV}(\alpha) = \emptyset$: The function $\mathrm{FV}(e)$ (where e can be θ, α, or ψ) gives us all the variables that can affect the behavior of an expression (the value of a term, the side effects of a program, or the truth of a formula), whereas $\mathrm{BV}(\alpha)$ gives us all the variables that can be modified during some program.

So for example a sound instance would be: $x = 0 \to [y := 1]x = 0$ but an unsound instance would be $x = 0 \to [x := 1]x = 0$.

Up into now we've always had axioms with preconditions (like $P \to Q$) and rules with premisses like:

$$\text{G}\,\frac{\phi}{[\alpha]\phi}$$

So what makes a side condition different vs. a premiss vs. a precondition? Premisses and preconditions are both $\mathsf{d}\mathcal{L}$ formulas, with the distinction that a premiss has to be true always (valid) while a precondition just has to be true right now. In contrast, what makes a side condition a side condition is that it's something we can't even write down in $\mathsf{d}\mathcal{L}$. Specifically, $\text{FV}(e)$ and $\text{BV}(\alpha)$ aren't $\mathsf{d}\mathcal{L}$. This has consequences for how we do a proof: because $\text{FV}(e)$ isn't a $\mathsf{d}\mathcal{L}$ formula, we're certainly not going to prove it ourselves in the middle of a $\mathsf{d}\mathcal{L}$ proof. The other two options are (a) get KeYmaera X to check it automatically for us and (b) give up and cry about how our prover is unsound. The good news is that $\text{FV}(e)$ and $\text{BV}(\alpha)$ are computable so we can get away with option (a).

And that's really where the question of prover complexity starts. We have a lot of axioms in $\mathsf{d}\mathcal{L}$ and for a lot of them it's not as obvious what the side conditions should be, for example if we're solving an ODE or adding a ghost variable. Not only are these side conditions non-obvious, but they lend themselves to lots of redundant error-checking code. Imagine how you would check whether the axiom schema V is applicable to a given formula ψ: you'd probably pattern match on ψ, look for something that looks like a $[\alpha]\phi$ and then check the side conditions on α and ϕ. It gets messy fast.

Uniform substitution is our solution to the mess. The first idea behind substitution is that axioms shouldn't be represented as *axiom schemata*, which is a fancy way of saying that there's a different axiom V for each α and ϕ, plus some code in the prover that figures out which α and ϕ apply for a given formula ψ. Instead an axiom should be a single concrete formula: this doesn't take any fancy code to implement; we can just write down a list of all our axioms in some file and BOOM we can use them now.

Extending dL With Uniform Substitutions The way we implement that is, surprisingly, by translating from Greek to Latin.[1] After translating, we have:

$$p() - > [a]p()$$

This might not look like anything serious, but what we really did was we extended the syntax of $\mathsf{d}\mathcal{L}$ with some new features: predicate symbols and program symbols. The full list of additions looks like:

[1]Not only is $\mathsf{d}\mathcal{L}$ a classical logic, it also has an appreciation for the classics. This must be why André writes test cases in Latin.

$$\begin{aligned}
\theta &::= \cdots \mid c() \mid f(x) \mid f(\bar{x}) \\
\alpha &::= \cdots \mid a \\
\phi &::= \cdots \mid p() \mid p(x) \mid p(\bar{x}) \mid C(\phi)
\end{aligned}$$

For terms θ we add functions $f(x)$, which can have any number of arguments. In the special case of zero arguments $c()$ we can also call c a constant (this, by the way, is why KeYmaera X adds all those parentheses to your constants now). In the case where all of the variables in the world can appear as arguments, we write $f(\bar{x})$. For programs there's no special need for arguments, so we just have program symbols a which stand for programs α. The story for formulas is more similar to terms with predicates $p()$ that can have any number of arguments, including all the arguments $p(\bar{x})$. We additionally have $C(\phi)$, called a predicational or quantifier symbol. Here C stands for a function from formulas to formulas, and as the name suggests, you can use it to insert quantifiers, like $C(\phi) = \forall x.\exists y.\phi$. You'll see this affects the semantics as well: For example instead of $[\![\theta]\!]_\omega$ we now have $[\![\theta]\!]_{I\omega}$. We added I, which is an *interpretation*: The interpretation tells what functions, predicates, etc. mean, similar to how the state ω tells us what all the variables mean. On the other hand, I doesn't change during a program because there aren't any programs that modify functions. For a formula to be valid it has to work for all interpretations. This is another way of saying that symbols are *uninterpreted*: when we write down a function $f(x)$ we can't assume that f means any particular function. If this doesn't sound exciting to you, you can ignore the interpretations when we write down the semantics and everything should still make sense.

This is a step in the right direction because now the axiom V can be a real, live formula $p() \to [a]p()$. But it's not so great all by itself. At the end of the day we want to instantiate the axiom and get stuff like $x = 0 \to [y := 1]x = 0$ — those $p()$ and a are super boring.

The Uniform Substitution Rule *Uniform Substitution* is the proof rule that lets us turn symbols like $p()$ and a back into $x = 0$ and $y := 1$. Not only that, it allows us to take care of all our side conditions once and for all (soundly!), no matter how many axioms we want to add later. A substitution σ is a function that replaces symbols with expressions. For example the instance of V above would use the substitution $\sigma(p()) = (x = 0), \sigma(a) = (y := 1)$, which we generally write $\{p() \mapsto x = 0, a \mapsto y := 1\}$. What makes this a *uniform* subsitution is that for any given symbol like $p()$, the substitution σ needs to replace it with the same thing everywhere. This might seem obvious but is actually super important: if I replaced the first $p()$ with $x = 0$ and the second with $x = 1$ then I'd get bogus stuff like $x = 0 \to [y := 1]x = 1$. The uniform substitution proof rule itself looks like this:

$$\text{US}\,\frac{\phi}{\sigma(\phi)}$$

That says that if we take any valid formula and apply a substitution to it, we'll get back another valid formula. Since axioms like V are all valid formulas, this gives us a way to instantiate them. But at the same time, an instance of V is only a valid formula if $\mathrm{FV}(\phi) \cap \mathrm{BV}(\alpha) = \emptyset$. The US rule is going to need a side condition that not only captures this, but also every side condition for every formula ever. Wow that sounds like a lot of work! And indeed that's why the side condition for US (called admissibility) is nontrivial:

Definition (Admissibility)**.** A substitution σ is admissible for ϕ iff $\mathrm{FV}(\sigma|_{\Sigma(e)}) \cap \mathrm{BV}(\otimes(\cdot)) = \emptyset$ for each operation $\otimes(e)$ in ϕ.

Let's tear the definition apart one piece at a time. "Each operation $\otimes(e)$" means we recursively check all operations in ϕ that bind variables. This can include formulas, specifically quantifiers like $\forall x.\phi$ that bind x, but also importantly includes programs; for example $x := \theta$ binds x and $x' = 1$ binds both x and x'. Intuitively $\mathrm{FV}(\sigma)$ means all the free variables introduced by the substitution, such as x in the casae of $x = 0$ and y in the case of $y := 1$ for our example substitution σ. The notation $\sigma|_{\Sigma(e)}$ says we don't have to worry about all of $\mathrm{FV}(\sigma)$, just the ones that will actually show up when we apply $\sigma(e)$ (here $\Sigma(e)$ is the *signature* of e, meaning all the symbols that appear in e).

3 Static Semantics

Clearly, the functions $\mathrm{BV}(\alpha), \mathrm{FV}(e)$, and Σ are essential to understanding soundness for US, so let's look at them closer. Moreover, they are interesting in their own right: together we call these functions the *static semantics* of $\mathsf{d}\mathcal{L}$ because they tell us something interesting about a $\mathsf{d}\mathcal{L}$ formula without having to run it, by contrast with the *dynamic semantics* which are all about what it means to run a program or formula. The static semantics of $\mathsf{d}\mathcal{L}$ are interesting in the same way as a type system is interesting for a programming language: a good static semantics should always allow us to decide whether a program "makes sense" without running it. For a program, "making sense" might mean that it doesn't segfault. In $\mathsf{d}\mathcal{L}$ we don't have segfaults and in fact the only type we have is the reals so any type system we could dream up would be boring. Instead our semantics focus on another important aspect for imperative programming: when is a variable an input or an output? Indeed even when we're not worrying about uniform susbstitutions, thinking about a $\mathsf{d}\mathcal{L}$ program in terms of inputs and outputs can clarify our thinking.

The *free variables* of an expression e, $\mathrm{FV}(e)$, are the inputs of an expression. The *signature* $\Sigma(e)$ is like $\mathrm{FV}(e)$ except it gives us all the symbols (functions, predicates, etc.) that affect e, not variables. The *bound variables* of a program α are the outputs (the same variable can be both input and output). We have some nice semantics lemmas that describe what it means to be an input or output. Below we say $\omega = \nu$ on S if for all variables $x \in S, \omega(x) = \nu(x)$:

Lemma (Bound Effect)**.** *Only outputs change. If $(\omega, \nu) \in [\![\alpha]\!]_I$ then $\omega = \nu$ on $BV(\alpha)^C$.*

Lemma (Coincidence)**.** *Only inputs (signature and free variables) affect the result of an expression. If $I = J$ on $\Sigma(e)$*

- *If $\omega_1 = \omega_2$ on $FV(\theta)$ then $[\![\theta]\!]_{I\omega_1} = [\![\theta]\!]_{J\omega_2}$.*
- *If $\omega_1 = \omega_2$ on $FV(\phi)$ then $[\![\phi]\!]_I = [\![\phi]\!]_J$.*
- *If $\omega_1 = \omega_2$ on $V \supseteq FV(\alpha)$ and $(\omega_1, \nu_1) \in [\![\alpha]\!]_I, (\omega_2, \nu_2) \in [\![\alpha]\!]_J$ then $\nu_1 = \nu_2$ on $V \cup MBV(\alpha)$.*

In the program case of coincidence, the "results" of a program are the final values of the variables, so coincidence says that variables which started out the same stay that way. Furthermore, we have yet another static semantic function: the *must-bound* variables $\mathrm{MBV}(\alpha)$ which are variables that get bound on every execution path of α no matter what. If a variable gets bound on every path then it can depend only on the free variables, so coincidence additionally tells us those variables agree in the end states.

3.1 Implementing Static Semantics

All the static semantic functions are implemented by recursion on expressions, but some definitions are interesting! The propositional cases just collect free variables from subformulas:

$$\mathrm{FV}(\phi \wedge \psi) = \mathrm{FV}(\phi) \cup \mathrm{FV}(\psi)$$

A quantifier $\forall x \phi$ overwrites x so x is no longer free in ϕ

$$\mathrm{FV}(\forall x \phi) = \mathrm{FV}(\phi)\ \{x\}$$

When we've got a program in a modality, we always depend on the inputs of the program, and we'll also depend on the inputs to ϕ, except the ones that we got from running α:

$$\mathrm{FV}([\alpha]\phi) = \mathrm{FV}(\alpha) \cup (\mathrm{FV}(\phi) \setminus \mathrm{MBV}(\alpha))$$

Exercise: Why is this alternate definition incorrect?:

$$\mathrm{FV}([\alpha]\phi) = (\mathrm{FV}(\alpha) \cup \mathrm{FV}(\phi)) \setminus \mathrm{MBV}(\alpha)$$

Program symbols stand for arbitrary programs, which means they might stand for a program that reads any variable we could ever possible imagine. To capture this possibility, we make every variable a free variable of a program symbol:

$$\mathrm{FV}(a) = V \cup V'$$

ODEs definitely depend on the RHS of the ODE and the domain constraint, but don't forget that the final value of x is also going to depend on the initial value:

$$\mathrm{FV}(x' = f(x) \& Q) = \{x\} \cup \mathrm{FV}(f(x)) \cup \mathrm{FV}(Q)$$

The story is similar for bound variables, but if you look at the bound effect theorem you'll notice it only talks about bound variables for programs. We don't have such a nice theorem for the bound variables of formulas and in fact the concept is not as important for formulas. An interesting case for programs is the ODE case:

$$\mathrm{BV}(x' = f(x) \& Q) = \{x, x'\}$$

Don't forget that an ODE always modifies both x and its derivative x'. It's important that we update x', otherwise if we had a differential term (θ) in the postcondition of an ODE, it wouldn't make a lot of sense.

Must-bound variables are similar to bound variables, but we take intersections instead of unions when there's a nondeterministic choice (except ODEs still bind variables even at duration 0):

$$\begin{aligned}
\mathrm{MBV}(\alpha \cup \beta) &= \mathrm{MBV}(\alpha) \cap \mathrm{MBV}(\beta) \\
\mathrm{MBV}(\alpha; \beta) &= \mathrm{MBV}(\alpha) \cup \mathrm{MBV}(\beta) \\
\mathrm{MBV}(x' = f(x) \& Q) &= \{x, x'\} \\
\mathrm{MBV}(\alpha^*) &= \emptyset
\end{aligned}$$

4 Deriving Axiom Schemata with US

I boldly claimed that US was powerful enough to implement V, now it's time to show that. If you give me a formula ψ and you want to apply V, here's what I do. I still have to break it down into a find out what α and ϕ are supposed to be. The good news is there's also an algorithm that can do that once-and-for called *unification*. We won't go into it for the sake of time, but the important part is that this step can't affect soundness at all. What we're doing is trying to come up with a substitution σ to plug into US, for example $\sigma \equiv \{p() \mapsto x = 0, a \mapsto y := 1\}$. If we mess up here, US will tell us that the substitution was bad and avert any real crisis. Once we've got σ, we can plug it in to the axiom V with US to get back our favorite instance of V:

$$\text{US}\,\dfrac{\text{V}\,\dfrac{*}{p() \rightarrow [a]p()}}{\sigma(p() \rightarrow [a]p())}$$

But this only works if the admissibility side condition is satisfied. It's instructive to see what admissibiity comes down to if we plug in a specific axiom like V for ϕ:

Definition (Admissibility for V)**.** A substitution $\sigma \equiv \{p() \mapsto \phi, a \mapsto \alpha\}$ is admissible for V iff $\mathrm{FV}(\phi) \cap \mathrm{BV}(\alpha) = \emptyset$.

How'd we get this definition? There's only one "operator" that can bind variables in V: the box modality $[a]p$ ($\rightarrow$ is also an operator, but does not bind variables). The bound variables of $[a]\phi$ are just the bound variables of a. Here the "argument" e is the proposition p, so we look at $\Sigma(p)$ which is just p itself, then $\sigma|_p$ is the substitution $\sigma' \equiv \{p() \mapsto \phi\}$ and we have $\mathrm{FV}(\sigma') = \phi$.

The moral of the story? The side condition for the V axiom and the side condition for US are saying the same thing, except US is saying it its full, general glory while V is giving it just in one very specific case.

For another example, consider the assignment axiom: $p(f()) \leftrightarrow [x := f()]p(x)$ and the instance $y^2 > 0 \leftrightarrow [x := y]x^2$ that we get with the substitution $\sigma \equiv \{f() \mapsto y, p(z) \mapsto z^2 > 0\}$. Here the bound variables of the box modality are again the bound variables of the program, which this time means just x. What are the free variables for $p(z) \mapsto z^2 > 0$, though? This is the first substitution we've seen for predicates that have arguments. These are not free variables, and in fact that's the whole reason for having functions and predicates with arguments in the first place. To emphasize this, we'll often write substitutions like $p(\cdot) \mapsto (\cdot)^2 > 0$ to emphasize that the argument $(\cdot)$ isn't treated like a variable, and especially not a free variable. However, when we substitute in $p(x)$ then in maybe x ought to be a free variable. In general, the arguments to functions and predicates should contribute to the free variables, because the result of a predicate certainly ought to depend on the arguments. However, they contribute the the free variables of $p(x)$, not the free variables of the substitution σ, which were what we cared about!

Clashes So far we've been looking at examples where everything goes nice and happy, but when a substitution goes bad, how do we detect that it was supposed to go bad? Here's an example of a dangerously bad substitution that clashes for the assignment axiom given above: $\sigma = \{f \mapsto x+1, p(y) \mapsto y \neq x\}$. This substitution is unsound because it gives us the following formula as an instance:

$$[x := x+1]x \neq x \leftrightarrow x + 1 \neq x$$

That's bogus. It should go without saying that $x = x$ and $x + 1 \neq x$ so the left-hand side is false and the right-hand side is true, making the equivalence quite false. How does US detect the clash? Here the replacement for $p(y)$ has x as a free variable, *regardless* of whether the argument is x or something else instead. This means that x will be a free variable of σ and even $\sigma|_{\Sigma(p(x),p(e)}$ where it wasn't in the last example. Then because x is also a bound variable of $x := 0$ we get $\mathrm{FV}(\sigma|_{\Sigma(p(x))}) \cap \mathrm{BV}(x := 0) = \{x\} \neq \emptyset$ and the US algorithm detects a clash. Moreover, why *should* we prohibit this substitution. Why was the result of this substitution invalid in the first place? The underlying intuition is right there in the name: this is supposed to be *uniform* substitution, meaning that if we replace $p(\cdot)$ with a formula ϕ, then that ϕ needs to mean the same thing everywhere. Of course it's allowed to vary by the argument *cdot* because that's what it means to be an argument, but any other variables we use have to stay the same. And our definition for p talks about x (i.e. introduces it as a free variable), so this substitution is only going to be kosher if x means the same thing everywhere. What's the simplest way to say that a variable means the same thing everywhere? By saying it never gets bound by any formulas or programs, because in the value of x everywhere agrees with (specifically) its initial value.

This principle is violated in the clashing substitution. On the left-hand side, x appears under an assignment $x := x + 1$. If we let x_0 be the initial value and x_1 be the value after the assignment, then on the left-hand side the x is x_1 but on the RHS it's x_0. If they were both x_0 or both x_1 we'd be fine, which would correspond to the valid formulas $[x := x + 1]x \neq x - 1 \leftrightarrow x + 1 \neq x$ or $[x := x + 1]x \neq x \leftrightarrow x + 1 \neq x + 1$ respectively. So in summary, we should be careful when we see programs that modify variables: if we see the same variable x before and after a program runs, it's essential for us to think about whether those two x's are still the same!

5 Applications