This chapter is heavily under construction

Tactics and Metaprogramming with Meta-F*

So far, we have mostly relied on the SMT solver to do proofs in F*. This works rather well: we got this far, after all! However, sometimes, the SMT solver is really not able to complete our proof, or takes too long to do so, or is not *robust* (i.e. works or fails due to seemingly insignificant changes).

This is what Meta-F* was originally designed for. It provides the programmer with more control on how to break down a proof and guide the SMT solver through the proper path via using *tactics*. Moreover, a proof can be fully completed within Meta-F* without using the SMT solver at all! This is the usual approach taken in other proof assistants (such as Lean, Coq, or Agda), but it's not the preferred route.

Meta-F* also allows for *metaprogramming*, i.e. generating programs (or types, or proofs ...) automatically. This should not be surprinsing to anyone already familiar with proof assistants and the Curry-Howard correspondence. There are however some slight differences between the two approaches, and more so in F*, so we will first look at automating proofs (i.e. tactics), and then turn to metaprogramming (though we use the generic name "metaprogram" for tactics as well).

In summary, when the SMT "just works", then we usually do not bother writing tactics, but we still have the ability to roll up our sleeves and write explicit proofs.

Speaking of rolling up our sleeves, let us do just that.

Decorating assertions with tactics

As you know already, F* verifies programs by computing verification conditions (VCs) and calling an SMT solver (Z3) to prove them. Most simple proof obligations are handled completely automatically by Z3, and for more complex statements we can help the solver find a proof via lemma calls and intermediate assertions. Even when using lemma calls and assertions, the VC for a definition is sent to Z3 in one single piece. This "monolithic" style of proof can become unwieldy rapidly, particularly when the solver is being pushed to its limits.

The first ability Meta-F* provides is allowing to attach tag specific tactics of assertions. These tactics operate on the "goal" that we want to prove, and can "massage" the assertion by simplifying it, splitting into more parts, tweaking particular SMT options, etc.

For instance, let us take the the following example, where we want to guarantee that `pow2 x` is less than one million given that `x` is at most `19`. One way of going about this proof is by noting that `pow2` is an increasing function, and that `pow2 19` is less than one million, so we try to write something like this:

```
let pow2_bound_19 (x:nat{x <= 19}) : Lemma (pow2 x < 1000000) =
  assert (forall (x y : nat). x <= y ==> pow2 x <= pow2 y);
  assert (pow2 19 ==  524288);
  assert (pow2 x < 1000000);
  ()
```

Sadly, this doesn't work. First of all, Z3 cannot automatically prove that `pow2` is increasing, but that is to be expected. We could prove this by a straightforward induction. However, we only need this fact for `x` and `19`, so we can simply call `FStar.Math.Lemmas.pow2_le_compat` from the library:

```
let pow2_bound_19' (x:nat{x <= 19}) : Lemma (pow2 x < 1000000) =
  FStar.Math.Lemmas.pow2_le_compat 19 x;
  assert (pow2 19 == 524288);
  assert (pow2 x < 1000000);
  ()
```

Now, the second assertion fails. Z3 will not, with the default fuel limits, unfold pow2 enough times to compute `pow2 19` precisely. Here we will use our first call into Meta-F*: via the `by` keyword, we can attach a tactic to an assertion. In this case, we'll ask Meta-F* to `compute()` over the goal, simplifying as much as it can via F*'s normalizer, like this:

```
let pow2_bound_19'' (x:nat{x <= 19}) : Lemma (pow2 x < 1000000) =
  FStar.Math.Lemmas.pow2_le_compat 19 x;
  assert (pow2 19 == 524288) by compute ();
  assert (pow2 x < 1000000);
  ()
```

Now the lemma verifies! Meta-F* reduced the proof obligation into a trivial equality. Crucially, however, the `pow2 19 == 524288` shape is kept as-is in the postcondition of the assertion, so we can make use of it! If we were just to rewrite the assertion into `524288 == 524288` that would not be useful at all.

How can we know what Meta-F* is doing? We can use the `dump` tactic to print the state of the proof after the call to `compute()`.

```
let pow2_bound_19''' (x:nat{x <= 19}) : Lemma (pow2 x < 1000000) =
  FStar.Math.Lemmas.pow2_le_compat 19 x;
  assert (pow2 19 == 524288) by (compute (); dump "after compute");
  assert (pow2 x < 1000000);
  ()
```

With this version, you should see something like:

```
Goal 1/1
x: x: nat{x < 20}
p: pure_post unit
uu___: forall (pure_result: unit). pow2 x < 1000000 ==> p pure_result
pure_result: unit
uu___'0: pow2 x <= pow2 19
------------------------------------------------------------------------------
squash (524288 == 524288)
(*?u144*) _
```

as output from F* (or in the goals buffer if you are using emacs). The `print` primitive can also be useful.

A "goal" is some proof obligation that is yet to be solved. Meta-F* allows you to capture goals (e.g. via `assert..by`), modify them (such as with `compute`), and even to completely solve them. In this case, we can solve the goal (without Z3!) by calling `trivial()`, a helper tactic that discharges trivial goals (such as trivial equalities).

```
let pow2_bound_19'''' (x:nat{x <= 19}) : Lemma (pow2 x < 1000000) =
  FStar.Math.Lemmas.pow2_le_compat 19 x;
  assert (pow2 19 == 524288) by (
    compute ();
    trivial ();
    qed ()
   );
  assert (pow2 x < 1000000);
  ()
```

If you `dump` the state just after the `trivial()` call, you should see no more goals remain (this is what `qed()` checks).

> **Note**
>
> Meta-F* does not yet allow an interactive style of proof, and hence we need to re-check the entire proof after every edit. This will be improved upon.

There is still the “rest” of the proof, namely that `pow2 x < 1000000` given the hypothesis and the fact that the assertion holds. We call this *skeleton* of the proof, and it is (by default) not handled by Meta-F*. In general, we only use tactics on those assertions that are particularly hard for the SMT solver, but leave all the rest to it.

The `Tac` effect

What, concretely, are tactics? So far we've written a few simple ones, without too much attention to their structure.

Tactics and metaprograms in F* are really just F* terms, but *in a particular effect*, namely `Tac`. To construct interesting metaprograms, we have to use the set of *primitives* provided by Meta-F*. Their full list is in the `FStar.Tactics.Builtins` module. So far, we have actually not used any primitive directly, but only *derived* metaprograms present in the standard library.

Internally, `Tac` is implemented via a combination of 1) a state monad, over a `proofstate`, 2) exceptions and 3) divergence. The state monad is used to implicitly carry the proofstate, without us manually having to handle all goals explicitly. Exceptions are a useful way of doing error handling. Any declared exception can be `raise`'d within a metaprogram, and the `try..with` construct works exactly as for normal programs. There are also `fail`, `catch` and `recover` primitives.

Metaprograms cannot be run directly. This is needed to retain the soundness of pure computations, in the same way that stateful and exception-raising computations are isolated from the `Pure` fragment (and from each other!). Metaprograms can only be used where F* expects them , such as in an `assert..by` construct. Here, F* will run the metaprogram on an initial proofstate consisting (usually) of a single goal, and allow the metaprogram to modify it.

To guarantee soundness, i.e. that metaprograms do not prove false things, all of the primitives are designed to perform small and correct modifications of the goals. Any metaprogram constructed from them cannot do anything to the proofstate (which is abstract) except modifying it via the primitives.

Having divergence as part of the `Tac` effect may seem a bit odd, since allowing for diverging terms usually implies that one can form a proof of false, via a non-well-founded recursion. However, we should note that this possible divergence happens at the *meta* level. If we call a divergent tactic, F* will loop forever waiting for it to finish, never actually accepting the assertion being checked.

As you know, F* already has exceptions and divergence. All `Dv` and `Ex` functions can readily be used in Meta-F* metaprograms, as well as all `Tot` and `Pure` functions. For instance, you can use all of the `FStar.List.Tot` module if your metaprogram uses lists.

Goals

Essentially, a Meta-F* tactic manipulates a *proofstate*, which is essentially a set of *goals*. Tactic primitives usually work on the goals, for example by simplifying (like `compute()`) or by breaking them down into smaller *sub*-goals.

When proving assertions, all of our goals will be of the shape `squash phi`, where `phi` is some logical formula we must prove. One way to break down a goal into subparts is by using the `mapply` tactic, which attempts to prove the goal by instantiating the given lemma or function, perhaps adding subgoals for the hypothesis and arguments of the lemma. This "working backwards" style is very common in tactics frameworks.

For instance, we could have proved the assertion that `pow2 x <= pow2 19` in the following way:

```
assert (pow2 x <= pow2 19) by (mapply (`FStar.Math.Lemmas.pow2_le_compat));
```

This reduces the proof of `pow2 x <= pow2 19` to `x <= 19` (the precondition of the lemma), which is trivially provably by Z3 in this context. Note that we do not have to provide the arguments to the lemma: they are inferred by F* through *unification*. In a nutshell, this means F* finds there is an obvious instantiation of the arguments to make the postcondition of the lemma and the current assertion coincide. When some argument is *not* found via unification, Meta-F* will present a new goal for it.

This style of proof is more *surgical* than the one above, since the proof that `pow2 x <= pow2 19` does not "leak" into the rest of the function. If the proof of this assertion required several auxiliary lemmas, or a tweak to the solver's options, etc, this kind of style can pay off in robustness.

Most tactics works on the *current* goal, which is the first one in the proofstate. When a tactic reduces a goal `g` into `g1,...,gn`, the new `g1,..,gn` will (usually) be added to the beginning of the list of goals.

In the following simplified example, we are looking to prove `s` from `p` given some lemmas. The first thing we do is apply the `qr_s` lemma, which gives us two subgoals, for `q` and `r` respectively. We then need proceed to solve the first goal for `q`. In order to isolate the proofs of both goals, we can `focus` on the current goal making all others temporarily invisible. To prove `q`, we then just use the `p_q` lemma and obtain a subgoal for `p`. This one we will just just leave to the SMT solver, hence we call `smt()` to move it to the list of SMT goals. We prove `r` similarly, using `p_r`.

```
assume val p : prop
assume val q : prop
assume val r : prop
assume val s : prop

assume val p_q : unit -> Lemma (requires p) (ensures q)
assume val p_q : squash p -> Lemma r
assume val qr_s : unit -> Lemma (q ==> r ==> s)

let test () : Lemma (requires p) (ensures s) =
  assert s by (
    mapply (`qr_s);
    focus (fun () ->
      mapply (`p_q);
      smt());
    focus (fun () ->
      mapply (`p_r);
      smt());
    ()
  )
```

Once this tactic runs, we are left with SMT goals to prove `p`, which Z3 discharges immediately.

Note that `mapply` works with lemmas that ensure an implication, or that have a precondition (`requires` / `ensures`), and even those that a squashed proof as argument. Internally, `mapply` is implemented via the `apply_lemma` and `apply` primitives, but ideally you should not need to use them directly.

Note, also, that the proofs of each part are completely isolated from each other. It is also possible to prove `p_gives_s` lemma by calling the sublemmas directly, and/or adding SMT patterns. While that style of proof works, it can quickly become unwieldy.

Quotations

In the last few examples, you might have noted the backticks, such as in `(`FStar.Math.Lemmas.pow2_le_compat)`. This is a *quotation*: it represents the *syntax* for this lemma instead of the lemma itself. It is called a quotation since the idea is analogous to the word “sun” being syntax representing the sun.

A quotation always has type `term`, an abstract type representing the AST of F*.

Meta-F* also provides *antiquotations*, which are a convenient way of modifying an existing term. For instance, if `t` is a term, we can write `` `(1 + `#t) `` to form the syntax of “adding 1” to `t`. The part inside the antiquotation (`` `# ``) can be anything of type `term`.

Many metaprogramming primitives, however, do take a `term` as an argument to use it in proof, like `apply_lemma` does. In this case, the primitives will will typecheck the term in order to use it proofs (though other primitives, such as `term_to_string`, won’t typecheck anything).

We will see ahead that quotations are just a convenient way of constructing syntax, instead of doing it step by step via `pack`.

Basic logic

Meta-F* provides some predefined tactics to handle "logical" goals.

For instance, to prove an implication `p ==> q`, we can "introduce" the hypothesis via `implies_intro` to obtain instead a goal for `q` in a context that assumes `p`.

Other basic logical tactics include:

- `forall_intro`: for a goal `forall x. p`, introduce a fresh `x` into the context and present a goal for `p`.
- `l_intros`: introduce both implications and foralls as much as possible.
- `split`: split a conjunction (`p /\ q`) into two goals
- `left` / `right`: prove a disjunction `p \/ q` by proving `p` or `q`
- `assumption`: prove the goal from a hypothesis in the context.
- `pose_lemma`: given a term `t` representing a lemma call, add its postcondition to the context. If the lemma has a precondition, it is presented as a separate goal.

(For experts: in Coq and other provers, this tactic is simply called `intro` and creates a lambda abstraction. In F* this is slightly more contrived due to squashed types, hence the need for an `implies_intro` different from the `intro`, explained ahead, that introduces a binder.)

See the `FStar.Tactics.Logic` module for more.

Normalizing and unfolding

We have previously seen `compute()`, which blasts a goal with F*'s normalizer to reduce it into a *normal form*. We sometimes need a bit more control than that, and hence there are several tactics to normalize goals in different ways. Most of them are immplemented via a few configurable primitives (you can look up their definitions in the standard library!)

- `compute()`: calls the normalizer with almost all steps enabled
- `simpl()`: simplifies logical operations (e.g. reduces `p /\ True` to `p`).
- `whnf()` (short for "weak head normal form"): reduces the goal until its "head" is evident.
- `unfold_def `t`: unfolds the definition of the name `t` in the goal, fully normalizing its body.
- `trivial()`: if the goal is trivial after normalization and simplification, solve it.

The `norm` primitive provides fine-grained control. Its type is `list norm_step -> Tac unit`. The full list of `norm_step` s can be found in the `FStar.Pervasives` module, and it is the same one available for the `norm` marker in `Pervasives` (beware of the name clash!).

Inspecting and building syntax

As part of automating proofs, we often need to inspect the syntax of the goal and the hypotheses in the context to decide what to do. For instance, instead of blindly trying to apply the `split` tactic (and recovering if it fails), we could instead look at the *shape* of the goal and apply `split` only if the goal has the shape `p1 /\ p2`.

Note: inspecting syntax is, perhaps obviously, not something we can just do everywhere. If a function was allowed to inspect the syntax of its argument, it could behave differently on `1+2` and `3`, which is of course bad! So, for the most part, we cannot simply turn a value of type `a` into a its syntax. Hence, quotations are *static*, they simply represent the syntax of a term, cannot turn values into terms. There is a more powerful mechanism of *dynamic quotations* that will be explained later, but suffice it to say for now that this can only be done in the `Tac` effect.

As an example, the `cur_goal()` tactic will return a value of type `typ` (an alias for `term`) representing the syntax of the current goal.

The `term` type is *abstract*: it has no structure in of itself. Think of it as an opaque "box" containing a term inside. A priori, all that can be done with a `term` is pass it to primitives that expect one, such as `tc` to type-check it `norm_term` to normalize it. But none of those give us full, programatic access to the structure of the term.

That's where the `term_view` comes in: following a classic idea in programming languages, there is function called `inspect` that turns a `term` into a `term_view`. The `term_view` type resembles an AST, but crucially is not recursive: it has `term` s (and not `term_view` s) where the subterms are.

Part of the `term_view` type.

```
noeq
type term_view =
  | Tv_FVar   : v:fv -> term_view
  | Tv_App    : hd:term -> a:argv -> term_view
  | Tv_Abs    : bv:binder -> body:term -> term_view
  | Tv_Arrow  : bv:binder -> c:comp -> term_view
  ...
```

The `inspect` primitves "peels away" one level of the abstraction layer, giving access to the top-level shape of the term.

The `Tv_FVar` node above represents (an ocurrence of) a global name. The `fv` type is also abstract, and can be viewed as a `name` (which is just `list string`) via `inspect_fv`.

For instance, if we were to inspect `` `qr_s `` (which we used above) we would obtain a `Tv_FVar v`, where `inspect_fv v` is something like `["Path"; "To"; "Module"; "qr_s"]`, that is, an "exploded" representation of the fully-qualified name `Path.To.Module.qr-s`.

Every syntactic construct (terms, free variables, bound variables, binders, computation types, etc) is modelled abstract like `term` and `fv`, and have corresponding inspection functions. A list can be found in `FStar.Reflection.Builtins`.

If the inspected term is an application, `inspect` will return a `Tv_App f a` node. Here `f` is a `term`, so if we want to know its structure we must recursively call `inspect` on it. The `a` part is an *argument*, consisting of a `term` and an argument qualifier (`aqualv`). The qualifier specifies if the application is implicit or explicit.

Of course, in the case of a nested application such as `f x y`, this is nested as `(f x) y`, so inspecting it would return a `Tv_App` node containing `f x` and `y` (with a `Q_Explicit` qualifier). There are some helper functions defined to make inspecting applications easier, like `collect_app`, which decompose a term into its "head" and all of the arguments the head is applied to.

Now, kwowing this, we would then like a function to check if the goal is a conjunction. Naively, we need to inspect the goal to check that it is of the shape `squash ((/\) a1 a2)`, that is, an application with two arguments where the head is the symbol for a conjunction, i.e. `(/\)`. This can already be done with the `term_view`, but is quite inconvenient due to there being *too much* information in it.

Meta-F* therefore provides another type, `formula`, to represent logical formulas more directly. Hence it suffices for us to call `term_as_formula` and match on the result, like so:

```
(* Check if a given term is a conjunction, via term_as_formula. *)
let isconj_t (t:term) : Tac bool =
  match term_as_formula t with
  | And _ _ -> true
  | _ -> false

(* Check if the goal is a conjunction. *)
let isconj () : Tac bool = isconj_t (cur_goal ())
```

The `term_as_formula` function, and all others that work on syntax, are defined in "userspace" (that is, as library tactics/metaprograms) by using `inspect`.

Part of the `formula` type.

```
noeq
type formula =
  | True_  : formula
  | False_ : formula
  | And    : term -> term -> formula
  | Or     : term -> term -> formula
  | Not    : term -> formula
  | Implies: term -> term -> formula
  | Forall : bv -> term -> formula
  ...
```

Note

For experts: F* terms are (internally) represented with a locally-nameless representation, meaning that variables do not have a name under binders, but a de Bruijn index instead. While this has many advantages, it is likely to be counterproductive when doing tactics and metaprogramming, hence `inspect` *opens* variables when it traverses a binder, transforming the term into a fully-named representation. This is why `inspect` it is effectul: it requires freshness to avoid name clashses. If you prefer to work with a locally-nameless representation, and avoid the effect label, you can use `inspect_ln` instead (which will return `Tv_BVar` nodes instead of `Tv_Var` ones).

Dually, a `term_view` can be transformed into a `term` via the `pack` primitive, in order to build the syntax of any term. However, it is usually more comfortable to use antiquotations (see above) for building terms.

Usual gotchas

- The `smt` tactic does *not* immediately call the SMT solver. It merely places the current goal into the "SMT Goal" list, all of which are sent to the solver when the tactic invocation finishes. If any of these fail, there is currently no way to "try again".
- If a tactic is natively compiled and loaded as a plugin, editing its source file may not have any effect (it depends on the build system). You should recompile the tactic, or just delete its object file to run it via the interpreter temporarily.

Coming soon

- Metaprogramming
- Meta arguments and typeclasses
- Plugins (efficient tactics and metaprograms, `--codegen Plugin` and `--load`)
- Tweaking the SMT options
- Automated coercions inspect/pack
- `e <: C by ...`
- Tactics can be used as steps of calc proofs.
- Solving implicits (Steel)