The Meta-theory of Symmetric Metaprogramming

Edit this page on GitHub

This note presents a simplified variant of principled metaprogramming and sketches its soundness proof. The variant treats only dialogues between two stages. A program can have quotes which can contain splices (which can contain quotes, which can contain splices, and so on). Or the program could start with a splice with embedded quotes. The essential restriction is that (1) a term can contain top-level quotes or top-level splices, but not both, and (2) quotes cannot appear directly inside quotes and splices cannot appear directly inside splices. In other words, the universe is restricted to two phases only.

Under this restriction we can simplify the typing rules so that there are always exactly two environments instead of having a stack of environments. The variant presented here differs from the full calculus also in that we replace evaluation contexts with contextual typing rules. While this is more verbose, it makes it easier to set up the meta theory.

Syntax

```
Terms         t  ::=  x                  variable
                      (x: T) => t        lambda
                      t t                application
                      ’t                 quote
                      ~t                 splice

Simple terms  u  ::=  x  |  (x: T) => u  |  u u

Values        v  ::=  (x: T) => t        lambda
                      ’u                 quoted value
```

```
Types          T  ::=  A                    base type
                       T -> T               function type
                       'T                   quoted type
```

Operational semantics

Evaluation

```
            ((x: T) => t) v  --> [x := v]t

                         t1  --> t2
                    ---------------
                       t1 t  --> t2 t

                         t1  --> t2
                    ---------------
                       v t1  --> v t2

                         t1  ==>  t2
                     -------------
                        't1  --> 't2
```

Splicing

```
                        ~'u  ==>  u

                         t1  ==>  t2
               -------------------------------
               (x: T) => t1  ==>  (x: T) => t2

                         t1  ==>  t2
                    ---------------
                       t1 t  ==>  t2 t

                         t1  ==>  t2
                    ---------------
                       u t1  ==>  u t2

                         t1  -->  t2
                     -------------
                        ~t1  ==>  ~t2
```

Typing Rules

Typing judgments are of the form `E1 * E2 |- t: T` where `E1, E2` are environments and `*` is one of `~` and `'`.

```
                  x: T in E2
               ---------------
               E1 * E2 |- x: T


           E1 * E2, x: T1 |- t: T2
       --------------------------------
       E1 * E2 |- (x: T1) => t: T -> T2


 E1 * E2 |- t1: T2 -> T    E1 * E2 |- t2: T2
 -------------------------------------------
              E1 * E2 |- t1 t2: T


               E2 ' E1 |- t: T
              -----------------
              E1 ~ E2 |- 't: 'T


               E2 ~ E1 |- t: 'T
               ----------------
               E1 ' E2 |- ~t: T
```

(Curiously, this looks a bit like a Christmas tree).

Soundness

The meta-theory typically requires mutual inductions over two judgments.

Progress Theorem

1. If `E1 ~ ⊢ t: T` then either `t = v` for some value `v` or `t ⟶ t2` for some term `t2`.
2. If `' E2 ⊢ t: T` then either `t = u` for some simple term `u` or `t ⟹ t2` for some term `t2`.

Proof by structural induction over terms.

To prove (1):

- the cases for variables, lambdas and applications are as in STLC.
- If `t = 't2`, then by inversion we have `' E1 ⊢ t2: T2` for some type `T2`. By the second induction hypothesis (I.H.), we have one of:
    - `t2 = u`, hence `'t2` is a value

- `t2 = u`, hence `t2` is a value;
- `t2 ⟹ t3`, hence `'t2 ⟶ 't3`.
- The case `t = ~t2` is not typable.

To prove (2):

- If `t = x` then `t` is a simple term.
- If `t = (x: T) ⇒ t2`, then either `t2` is a simple term, in which case `t` is as well. Or by the second I.H. `t2 ⟹ t3`, in which case `t ⟹ (x: T) ⇒ t3`.
- If `t = t1 t2` then one of three cases applies:
  - `t1` and `t2` are a simple term, then `t` is as well a simple term.
  - `t1` is not a simple term. Then by the second I.H., `t1 ⟹ t12`, hence `t ⟹ t12 t2`.
  - `t1` is a simple term but `t2` is not. Then by the second I.H. `t2 ⟹ t22`, hence `t ⟹ t1 t22`.
- The case `t = 't2` is not typable.
- If `t = ~t2` then by inversion we have `E2 ~ ⊢ t2: 'T2`, for some type `T2`. By the first I.H., we have one of
  - `t2 = v`. Since `t2: 'T2`, we must have `v = 'u`, for some simple term `u`, hence `t = ~'u`. By quote-splice reduction, `t ⟹ u`.
  - `t2 ⟶ t3`. Then by the context rule for `'t`, `t ⟹ 't3`.

Substitution Lemma

1. If `E1 ~ E2 ⊢ s: S` and `E1 ~ E2, x: S ⊢ t: T` then `E1 ~ E2 ⊢ [x := s]t: T`.
2. If `E1 ~ E2 ⊢ s: S` and `E2, x: S ' E1 ⊢ t: T` then `E2 ' E1 ⊢ [x := s]t: T`.

The proofs are by induction on typing derivations for `t`, analogous to the proof for STL (with (2) a bit simpler than (1) since we do not need to swap lambda bindings with the bound variable `x`). The arguments that link the two hypotheses are as follows.

To prove (1), let `t = 't1`. Then `T = 'T1` for some type `T1` and the last typing rule is

```
         E2, x: S ' E1 |- t1: T1
        -------------------------
        E1 ~ E2, x: S |- 't1: 'T1
```

By the second I.H. `E2 ' E1 ⊢ [x := s]t1: T1`. By typing, `E1 ~ E2 ⊢ '[x := s]t1: 'T1`. Since `[x := s]t = [x := s]('t1) = '[x := s]t1` we get `[x := s]t: 'T1`.

To prove (2), let `t = ~t1`. Then the last typing rule is

```
E1 ~ E2, x: S |- t1: 'T
-----------------------
E2, x: S ' E1 |- ~t1: T
```

By the first I.H., `E1 ~ E2 ⊢ [x := s]t1: 'T`. By typing, `E2 ' E1 ⊢ ~[x := s]t1: T`. Since `[x := s]t = [x := s](~t1) = ~[x := s]t1` we get `[x := s]t: T`.

Preservation Theorem

1. If `E1 ~ E2 ⊢ t1: T` and `t1 ⟶ t2` then `E1 ~ E2 ⊢ t2: T`.
2. If `E1 ' E2 ⊢ t1: T` and `t1 ⟹ t2` then `E1 ' E2 ⊢ t2: T`.

The proof is by structural induction on evaluation derivations. The proof of (1) is analogous to the proof for STL, using the substitution lemma for the beta reduction case, with the addition of reduction of quoted terms, which goes as follows:

- Assume the last rule was

  ```
    t1  ==>  t2
  -------------
   't1  --> 't2
  ```

  By inversion of typing rules, we must have `T = 'T1` for some type `T1` such that `t1: T1`. By the second I.H., `t2: T1`, hence `'t2:` T1`.

To prove (2):

- Assume the last rule was `~'u ⟹ u`. The typing proof of `~'u` must have the form

  ```
   E1 ' E2 |- u: T
  -----------------
  E1 ~ E2 |- 'u: 'T
  -----------------
  E1 ' E2 |- ~'u: T
  ```

  Hence, `E1 ' E2 ⊢ u: T`.

- Assume the last rule was

```
              t1  ==>  t2
      -------------------------------
      (x: S) => t1  ==>  (x: T) => t2
```

By typing inversion, `E1 ' E2, x: S ⊢ t1: T1` for some type `T1` such that `T = S → T1`. By the I.H, `t2: T1`. By the typing rule for lambdas the result follows.

- The context rules for applications are equally straightforward.
- Assume the last rule was

```
      t1  ==>  t2
     -------------
     ~t1  ==>  ~t2
```

By inversion of typing rules, we must have `t1: 'T`. By the first I.H., `t2: 'T`, hence `~t2: T`.

TASTy I... Other ...