Tips for using ATS Anairiats

BY S. CHANDRASEKARAN

Email: shiv@ece.ucsb.edu

September 1, 2011

ATS demands a lot more than other functional languages. Here I have assembled some tips that I mostly gathered by observing **Hongwei Xi** debug my programs. My hope is that this document will grow with contributions from all ATS users.

1. **Interchanging template type parameters and polymorphic type parameters**. Instead of `fun foo {a:nat} (...): void = ...` you type `fun {a:nat} foo (...): void = ...`. This can pass the type-checker and still fail to compile. Pay close attention to mixing these two types of parameters. Template parameters are only needed when the size of the type needs to be known in the code. When using templated code pass the template parameters explicitly: `goo<double> ()`, etc.

2. **Obvious type errors**. Sometimes the compiler will say some type is not right, even though you think the correct type can be derived. Sometimes this is because some of the polymorphic variables need to be supplied. Remember you need to enumerate them from left to right as they occur in their definition. For example this is frequently needed `array_ptr_initialize_clo_tsz {double} (pf | ...)`. That `{double}` specification is the key here.

3. **Something is not found**. There are three ways to include `.sats` (1) and `.dats` (2) files.

    1. `staload ‘‘fname.sats’’`

    2. `staload _(*anonymous*) = ‘‘fname.dats’’`

    3. `dynload ‘‘fname.dats’’`

    These are distinct forms. Sometimes you will need all three. If you need to use functions in `goo.sats` in `foo.dats`, then in `foo.dats` use the first form of `goo.sats`. The last two forms are *usually* only needed in the main program, say `main.dats`. If `goo.dats` contains full template implementations then you will use form 2 of `goo.dats` in `main.dats`. If the `goo` library require initialization then you will use form 3 of `goo.dats` in `main.dats`. If the compiler says that some template definition cannot be found, then you need form 2. If it complains about `dynload_flag` not being available instead then you need form 3. There is one more rule about partial template definitions. A *full* template definition looks like `implementation{a} foo {..} ... (...`, whereas a *partial* template definition looks like `implementation foo<T> {...} ... (...`. When you use a partial template definition then in the main program you need to include that definition as follows

    ```
    local (* empty *) in
    #include ‘‘goo.dats’’
    end
    ```

    You can put as many `#include`'s as needed in the local statement. Hongwei's latest suggestion is to do this as follows instead

    ```
    local #include ‘‘goo.dats’’ in (*empty*) end
    ```

    as this will prevent the contamination of the local namespaces with names from `goo`.

4. **.hat files**. If you have a `.dats` file with both full and partial templates it might be difficult to avoid multiple definitions. Move all the partial templates into a `.hats` file and `local in #include ''_.hats'' end` in the main program file. See previous comment for the new desired form of this statement instead to avoid namespace pollution.

5. **Closures**. Note that `cloptr1` does not require GC while `cloref1` does. The suffix 1 just indicates all effectrs are allowed. For example `:<cloptr,!ntm>` means it is a closure with only (potentially) non-terminating effect.

6. **Dynamic variables are not function arguments**. This error message might indicate that you have used variables in an internal function that has *not* been declared as a closure. Try `:<cloptr>` as the arrow decoration.

7. **Closures on stack**. To avoid GC and yet have higher-order functions you can allocate closures on stack. The syntax is as follows:

   ```
   var !p_foo = @lam (...) : int =<clo> ...
   ```

   However, I don't know how to make `!p_foo` a *recursive* function. In such cases one can declare `foo` as a `cloptr` instead:

   ```
   fun foo (...) :<cloptr1> int = ...
   ```

   In this case, ATS will use `malloc/free` to allocate and free the closure at the end of the function scope. If `malloc/free` are not available, then you have to use `@lam`. Note, `cloptr` does *not* need `GC`.

8. **View not available**. Sometimes when you use `array_ptr_initialize_clo_tsz` you will loose the view of some external arrays that are needed by the closure. If that occurs try adding the viewdef explicitly to the call `array_ptr_initialize_clo_tsz {double} {V} (...)` etc.. You might find it convenient to use `viewdef V = @[double][n] @ lx` or some such definition close-by. The same problem occurs when you try to use `GMAT_safe_mul` in `ATSfloat/nla.sats`. In this case the call should usually look like `GMAT_safe_mul<a> {v1,v2} (...` where `v1` and `v2` are usually defined as `viewdef v1 = GEMAT_v (a, m1, n1, ld1, l1)` and similarly for `v2`. The view variables are a *must* in this case.

9. **Uninitialized arrays.** These are hard to set-up yourself. See if you can use `array_ptr_initialize_clo_tsz`. Writing an obvious recursion will not work as the compiler will not be satsified that `@[double?][m] » @[double][m]` has truly been accomplished, especially during the recursion.

10. **Static initialization of arrays**. This is not quite clear from the manual and tutorial. The following syntax is an easy way to allocate and initialize an array:

    ```
    val (free_gc, pf_A | p_A, M) = $arrsz {double} (1.0, ~2.0, 3.14)
    ```

    where the type of `!p_A` is `@[double][3]` and `M` has type `size_t 3`. If you use the macro `array` or `matrix` you can get some simplifications of the returned type. See `array.sats` and `matrix.sats` for further information.

11. **Uninitialized fmatrix**. Sometimes you need to reference an entry you have already made in order to fill in another entry. It is hard to convince the compiler the previous entry is there when the type is of the form `a: &fmatrix(double?,...)`. The simplest approach is to

    - `fmatrix_ptr_initialize_elt<double> (..., a, ...)`

- `prval pf = view@ a` // change the default view
- work with `a` using the new implicit default `view@ a = fmatrix(double, ...)`
- `prval () = view@ a := pf` // reset the view

If a proof is already available explicitly with the uninitialized state you can update the master type with a trivial statement of the form:
`prval pf_arr = pf_arr`

Here is a *another* solution suggested by Hongwei Xi. Here is the code that will not work:

```
fun f (x: &int? >> int): void = let
  val () = x := 0
  val () = if x > 0 then x := 1
in
end
```

as the *master type* in the `if` is still `int?`. The new fix is:

```
fun f (x: &int? >> int): void = let
  val () = x := 0
  val () = if :(x: int) => x > 0 then x := 1
in
end
```

Note the change in syntax of the `if` statement. The old solution would be:

```
fun f (x: &int? >> int): void = let
  val () = x := 0
  prval pf = view@ x
  val () = if x > 0 then x := 1
  prval () = view@ x := pf
in
end
```

This works because the *master type* of `pf` is `int @ x`. For `for` statements master types can be introduced via the following syntax:

```
for* (pf_arr: array_v (int, N, l)) => (i := 0; i < N; i := i + 1)
  p_arr->[i] := 42
```

12. **Proofs and if statements**. Frequently one runs into the following situation:

```
pf: GMAT_v (a?, m, n, ld, l)
val () = if n > 1 then ... pf := GMAT (a, m, n, ld, l)
  else ... pf := GMAT (a, m, n, ld, l)
```

and yet after the `if` statement the type of `pf` will still be of the form `a?`. To fix this use a *state type* as follows
`val () = if :(pf: GMAT (a, m, n, ld, l)) => n > 1 then .. else`
Note that the type in the `if` statment is the type of `pf` *after* the `if` statement.

13. **Uninitialized array errors**. If you see the error message contains lines like: `S2Etyarr(S2Etop(0; S2Ecst(double))`, then you probably have an uninitialized array where an initialized array is required.

14. **Static variables**. Getting hold of a static variable can be tricky sometimes. Here are some common forms:

- ```
  stavar l: addr
  val _ = &x: ptr l // x: &type
  ```
  ```

- `val [mn:int] (pf_mn | mn) = m imul2 n`
- `val [n:int] n = int1_of (2 * x + y * z - a * a)`
- `val [l:addr] (pf_gc, pf_arr | p) = array_ptr_alloc<double> (n)`
- `var acc = 0 // implicit stavar acc: addr  val _ = &acc: ptr acc`

Perusing `libats/DATS/fmatrix.dats` is one of the best ways to get familiar with this.

15. **prval vs. val**. Sometimes type-checking will pass but you will get strange type errors when you go for the full compile. Look at the suspect line. Did you use `val` when you should have used `prval`? Sometimes you will get a message like `internal error: _the_dynctx_find_` and it will mention a prfun or something like that.

16. **Syntactically the same**. Sometimes two quantities that are the same will be rejected by the type-checker. For example minimum of two size types. In this case try to give the minimum a static name. For example

    - `starvar mn : int`
    - `val MN = min_size1_size1 (M, N) : size_t mn`

    Now `mn` is known to be equal to `min(m,n)` where `M : int m` and `N : int n`. Another way is to explicitly coerce the proof

    - `prval pf = pf : array_v (a, n, l)`

17. **Two identical static variables**. Sometimes you can define two static variables that are identical in the same scope and get strange error messages that two identical types don't have the same size.

18. **Unused for-all variable**. For example suppose if you have

    - `fn foo {a:int} {b:addr} (..`

    but actually you never use `b` anywhere. Then when you *call* `foo` the compiler will complain because it cannot fit `b`. So if you see a message like `no static variable fits the constraints`, look for the extra for-all variable.

19. **Datatypes and GC**. If a `datatype` has only unary constructors then it is represented internally as integers. So there is no need for GC in this case.

20. **extern vs. fun**. Sometimes you will get strange compile time errors (involving the key word `_dynctx_`) about a perfectly typed function. If that function is declared as `fn` and used in a template in the local file then you will get this error. The way out is to declare it as `extern` and then `implement` it.

21. **Array access**. Use `xs.[i]` in preference to `xs[i]`. You will get fewer type errors.

22. **scase vs. case**. This is more subtle than I thought. For `prfun {n:int} foo (pf: @[int][n] @ l): ...` , you have to use `scase` on `n`, but `case` on `pf`.

23. **Proof closures**. For these either use `llam` or even just `lam ... =<lin>` as `llam ... =<x>` is the same as `lam ... =<lin,x>`.

24. **Compiler crashes**. Okay, this is too generic, but if the type checker crashes, one possible reason could be that a view static variable needs to be explicitly stated. This will happen with functions that take arguments that are closures. For example, routines like `array_ptr_initialize_clo_tsz {a} {v}` and `GMAT_ptr_initialize_clo<a> {v}` require the view variable to be mentioned explicitly for the type checking to succeed without stack overflow.

25. **Records and tuples with views**. To define records with embedded proofs you can do `@{pf_arr = array_v (a, n, l), size = size_t n, ptr = ptr l}`, whereas with tuples you would do `@(array_v (a, n, l) | size_t n, ptr l)`. Note the lack of a vertical bar for records.

26. **Order of record fields**. When you create a record as `val a = @{m = 1, s = ‘‘He’’}: R`, the order of the fields `m` and `s` must follow the definition order for `viewtypedef R = @{m = int, s = string}`. This is to ensure that there is a fixed semantics for side-effects when the record is instantiated field-by-field.

27. **fold@ and free@**. These can only be used with `dataviewtype`'s; not with tuples and records. For tuples and records use their field names to access interior parts. For example `g.3->g.pf_gmat`.

28. **Accessing sub-fields of dataviewtypes**. This is essentially a cut-and-paste of an email I got from Hongwei. Suppose we define

```
datatype list_vt (a:type) =
  | list_vt_nil (a) of ()
  | list_vt_cons (a) of (a, list_vt (a))
```

and implement the function

```
fun{a:t@ype} length (xs: !list_vt (a)): int =
case xs of
| list_cons (x, xs1) => fold@ xs; 1 + length (xs1)
| list_nil () => fold@ xs; 0
```

then this will *not* type-check. We took out `xs1` from the cons cell but did not return it. You can fix it by using pointers as shown below.

```
fun{a:t@ype} length (xs: !list_vt (a)): int =
case xs of
| list_cons (x, !p_xs1) => fold@ xs; 1 + length (!p_xs1)
| list_nil () => fold@ xs; 0
```

or

```
fun{a:t@ype} length (xs: !list_vt (a)): int =
case xs of
| list_cons (x, !pxs1) => let
    val xs1 = !p_xs1       // get it out
    val n1 = length (xs1)
    val () = !p_xs1 := xs1 // return it
  in
    fold@ xs; n1+1
  end
| list_nil () => fold@ xs; 0
```

29. If you `malloc_gc` a block and then displace the pointer from the beginning of the block, GC might reclaim your block. If you need to do this, use `malloc_ngc` instead. Then, of course, you must reclaim the block yourself.

30. `#[n:int | n > 0]`. Very rarely, you will need to declare a function of the form

```
fun foo (..., x: &int >> int n, ...) :<> #[n:int | n > 0] =
  let ... in ... end
```

    Note that the static variable for `x` is used before being declared. The syntax `#[n...]` extends the scope of `n` correctly (for exact details you are going to have to ask Hongwei himself).

31. **Returning a static variable**. Say you have a function `fun foo (): [n:int] (int n)`. Sometimes the compiler will not be able to figure out the static variable for the return value. In that case you can add it explicitly yourself using the following syntax:

```
let ... in #[n | my_val_n] end
```

32. **Conflicting names and `staload`**. From 0.1.9 onwards if you open two `.sats` files with conflicting names `ATS` will complain. The way out is to give at least one of them a module name. As of now `number.sats` and `complex.sats` both have `abs` defined internally. So do `staload C = libc/SATS/complex.sats` to resolve the name conflict.

33. **Wrapping `C` libraries**. For simplicity, assume that the `C` code is in `foo.c` with headers in `foo.h`. You will create 4 ATS files: `foo.sats`, `foo.dats`, `foo.hats` and `foo.cats`.

    i. In `foo.cats` place any `C` wrappers you need so that it becomes easier to make the call from `ATS`. Make sure that the first two lines are

```
#ifndef ATSFOO
#define ATSFOO
```

    Then place the rest of your `C` code and make the last line

```
#endif
```

    This ensures that when the file `foo.cats` is included multiple times in the generated `C` code you don't generate compiler errors. Make sure to include the `C` header file in `foo.cats` with a statement of the form

```
#include ‘‘foo.h’’
```

    ii. In your `foo.sats` file include the `foo.cats` with the lines

```
%{#
#include ‘‘foo.cats’’
%}
```

    This is the way that `C` fragments are included in `.sats` files. This will also ensure that any types and `#define`'s that you need to introduce are available to you. Now define your ATS version of the `C` API.

    iii. If you need to define full template implementations of the form

```
implement{a} goo (a, b, c) = ...
```

    put them in `foo.dats`.

iv. If you need to define partial template implementations of the form

```
implement goo<double> (a, b, c) = ‘‘double_goo’’
```

put them in `foo.hats`.

v. Suppose now that your main program resides in `main.dats`. You can access your new ATS API via

```
staload ‘‘foo.sats’’
```

or, if you do not want to open the whole module and cause name clashes, via

```
staload F = ‘‘foo.sats’’
```

To allow access to the full template implementations you also need the line

```
staload _ = ‘‘foo.dats’’
```

To allow name safe access to the partial template implementation you also need the line

```
local #include foo.hats in (* empty *) end
```

Finally, in case your library needs run-time initialization you also need

```
dynload ‘‘foo.dats’’
```

vi. In your `Makefile` you will use a command that looks like

```
atscc foo.sats foo.dats main.dats -o foo -lfoo
```

vii. In the next tip I talk about controlling dynamic loading.

34. **Dynamic loading of** `.dats`. What happens if you don't need run-time initialization for `foo.dats`? If you skip `dynload ‘‘foo.dats’’` in `main.dats`, the ATS compiler will complain. The way out is to add the following line to `foo.dats`

```
#define ATS_DYNLOADFLAG 0 // dynamic load at run-time not needed
```

If you do this, please make sure that `foo.dats` only has functions and function templates. If you have a top-level expression of the form

```
val global_value = my_function ()
```

it will not be evaluated if you turn-off dynamic loading for this file. This will also prevent `global_value` from being marked as a root by the `gc`. So be cautious in turning off dynamic loading.

35. **Dynamic loading of** `.sats`. Strangely enough even a `.sats` file has to be compiled and loaded at run-time. To prevent this add the line

```
#define ATS_STALOADFLAG 0 // no staload at run-time
```

If you do this, please make sure that there are no `datatype`'s or `exception`'s defined in the `.sats` file.

36. `static inline` **for** `C` **glue**. When writing `C` code make sure to mark it as `static inline` to prevent multiple definitions:

```
static inline ats_size_type foo (ats_size_type n) {
  return (ats_size_type) (5 * n);
};
```

37. **Fake proofs from** `C` **functions**. A typical situation requires that the user provides a work-space array of `m` entries, if the data size is `n`, where `m` is related to `n` by some complicated function. If this relationship is not satisfied you might incur a run-time error. The way out is to define an abstract proposition encapsulating the relationship:

```
absprop MyProp (n:int, m:int)
```

Let `int comp_workspace_size (int n);` be the `C` function that computes the work-space size. You can declare the `C` function thus:

```
extern fun ats_comp_workspace_size {n:int} (n: int n):
  [m:int] @(MyProp (n,m) | int m)
  = ''comp_workspace_size''
```

Now `ats_comp_workspace_size` will return not only the correct workspace size, but also a certificate that the other functions can demand to see before they use the workspace of size `m` with the data of size `n`.

38. **Strange missing template compiler errors**. Sometimes the compiler will complain that a template instantiation for certain types is missing. Assuming you did not mess up how templates should be defined and loaded, look at the types that the compiler is trying to find a template for. If the types look strange it could be that you have forgotten to explicitly include the template type parameters in a template call somewhere. For example you might have `my_template (x, y)`, rather than `my_template<float, int> (x,y)`. The ATS compiler will insert template type parameters on its own, but sometimes they are not the ones for which you have instantiations. Adding the types explicitly will help.

39. **How to use** `list_vt`. There are 3 ways to pass `list_vt` arguments to functions:

    i. `fun foo (v: list_vt (...)) : void`. In this case the list is consumed after the function call and not available for later use. A typical example would be freeing the list.

    ii. `fun foo (v: !list_vt (...)) : void`. Here the pointer to the head of the list is passed in. So anything that it points to can be modified, but you will continue to have access to the same pointer to the head of the list.

    iii. `fun foo (v: &list_vt (...)) : void`. Here address of the pointer to the head of the list is passed in. So in principle the function could change even the pointer to the head of the list. In other words inside `foo` we can have a statement of the form `v := new_list_vt`. This would be illegal in either of the earlier forms. Note that in both of the earlier forms, the contents of any cons cell, like the tail for example, can be assigned. So there is no write-protection in the earlier forms.

Just remember that `v: list_vt`, mean that `v` is actually a pointer to the `head` struct of the list. As far as I can tell you probably can (should?) avoid version iii. For example, while the tutorial uses this form for `reverse_append`, `list_vt.sats` does not have a single instance of this!