# ATS Docs and Tips Documentation

***Release 0.0.1***

**ATS Community**

October 21, 2014

Contents

I hope this to be a place where all the people in the community could contribute docs and tips to boost ATS users.

Contents:

CHAPTER 1

# Installation

## 1.1 Install ATS 2 from Source Code

Before installing ATS 2, we need the latest version of ATS 1 first.

1. Download ATS 0.2.10 from http://sourceforge.net/projects/ats-lang/files/
2. Unzip it into a folder, e.g. `~/ats`
3. Setup environment variables

```
export ATSHOME=~/ats
export ATSHOMERELOC=ATS-0.2.10
export PATH=$PATH:$ATSHOME/bin
```

4. Configure

> **Attention:** You may need to install `autoconf` package first

```
cd ~/ats
aclocal
autoheader
automake --force-missing --add-missing
autoconf
./configure
```

5. Make

```
make all
```

6. Make sure that all the binaries are under `bin`, and all the libraries are under `ccomp/lib` and `ccomp/lib64`.

Next, we are building ATS 2. CMake 2.8+ is recommended.

1. Download source code from GitHub, https://github.com/githwxi/ATS-Postiats/archive/master.zip
2. Unzip it into a folder, e.g. `~/ats2`
3. Setting up environment variables

```
export ATSHOME=~/ats
export ATSHOMERELOC=ATS-0.2.10
export PATSHOME=~/ats2
export PATH=$PATH:$ATSHOME/bin:$PATSHOME/bin
```

4. Make

```
cd ~/ats2/src/BUILD
cmake ..
make

mkdir ~/ats2/bin
cp patsopt ~/ats2/bin/
```

Next, build libraries and tools

1. Generate source code from templates

```
cd ~/ats2
make -f codegen/Makefile_atslib
make -f codegen/Makefile_atscntrib
```

2. Make C Target Compiler and Libraries

```
cd ~/ats2/ccomp
make
```

3. Make utilities

```
cd ~/ats2/libatsyntax
make

cd ~/ats2/utils/atscc
make
cp patscc ~/ats2/bin/

cd ~/ats2/utils/atsyntax
make
```

CHAPTER 2

# Features

## 2.1 Function Effects

**Note:** This part is taken from Chris Double's blog and a discussion on ATS google group. Thanks to all the contributions in the blog and group thread.

For a function like `fun foo (someargs: sometype):<> sometype`, the `:<>` part is used to describe effects of functions. There is a sort `eff` exclusively for function effects. You can do things like `:<>`, `:<lin,prf>` and so on. The meaning of them are as follows.

**`:<>`** pure, no effects at all

**`!exn`** the function possibly raises exceptions

**`!ntm`** the function possibly is non-terminating

**`!ref`** the function possibly updates shared memory, which means reading from or writing to a location that it knows exists but does not own.

**`!wrt`** (ATS2 only) the function may write to a location it owns

**`0`** the function is pure (has no effects)

**`1`** the function can have all effects

**`fun`** the function is an ordinary, non-closure, function

**`clo`** the function is a stack allocated closure

**`cloptr`** the function is a linear closure that must be explicitly freed

**`cloref`** the function is a peristent closure that requires the garbage collector to be freed.

**`lin`** the function i slinear and can be called only once

**`prf`** the function is a proof function

## 2.2 Quantifiers

In ATS, `[]` is mostly used as extential quantifier, while `{}` is mostly used as universal quantifier.

For example, suppose `MUL` encodes the multiplication relationship as defined here, we can write something like this.

```
prfun mutiplication_is_total {m,n:int} (): [p:int] MUL (m, n, p)
```

which will be interpreted as

> For all integers *m* and *n*, there exists some integer *p* such that `MUL (m, n, p)` is ture.

CHAPTER 3

# Examples

## 3.1 Creating ATS Interface for C Library

I did a zlog interface for ATS based on Zhiqiang Ren's work. You can access the code at GitHub. And here's how I did it.

### 3.1.1 Getting Start

Download and install zlog, read its documents and interfaces, get familiar with it by writting some simple hello world in C.

### 3.1.2 Translate into a Minimum Workable ATS Interface

For a minimum hello world, we just write such a C program

```
int main () {
        zlog_init ("")
        zlog_category *c = zlog_get_category ("mycat");
        zlog (c, "%s", "hello world");
        zlog_fini ();
}
```

I consider the data structures first.

`zlog_category *` is used across interfaces, but I don't care it's inner structure, which means I can use an **abstract type**. It doesn't need to be freed manually, instead, all resources are freed by `zlog_fini ()`. Therefore I make it **non-linear** in ATS. It is a pointer in C, I then use **boxed type** in ATS.

Therefore, I can define `zlog_category` using `abstype`, which is abstract, non-linear, and boxed.

```
abstype zlog_category
```

For functions, we can translate them directly into ATS

```
fun zlog_init (config: string): int = "mac#zlog_init"
fun zlog_get_category (name: string): zlog_category = "mac#zlog_get_category"
fun zlog_fini (): void = "mac#zlog_fini"
```

The logging function `zlog` is tricky since it has variable length parameter list. What I do here is using an intermediate C function.

```
fun zlog {ts:types}
(c: zlog_category, level: int, fmt: string, args: ts): void =
"mac#zlog_handler"
```

And in the DATS file, I implement it as follows

```
#include "zlog.h"
#include <stdarg.h>

void zlog_handler (zlog_category *c, int level, char *fmt, ...) {
        va_list args;
        va_start (args, fmt);

        vzlog (
                c,
                __FILE__,
                sizeof(__FILE__) - 1,
                __func__,
                sizeof(__func__) - 1,
                __LINE__,
                level,
                fmt,
                args);

        va_end (args);
}
```

The `vzlog` is an alternative interface provided `zlog` itself. Most library interfaces will provide such an alternative besides its original variable length version.

And now we can use them to write the hello world example.

### 3.1.3 Refining Types

My initial type for `zlog_category *` is just workable. But now I wanna force programmers to check whether it is a null pointer before using it.

I then change the definition and interface to the followings.

CHAPTER 4

# Library Docs

CHAPTER 5

# TBD

## 5.1 At-View Sugar

`#view`, `#sugar`

```
fun {a:t@ype}
ptr_get {l:addr} (pf: !a @ l >> a @ l | p: ptr (l)): a
```

is actually a sugar for

```
fun {a:t@ype}
ptr_get {l:addr} (pf: a @ l | p: ptr (l)): (a @ l | a)
```

The de-sugar one says everything about the meaning of `!` and >>.

foo (pf: !T?@l >> T@l | ptr (l))

@[T][n] Array

---

The #[...] in existential qualifiers means that that variable can be referenced in function parameters. Notice the 'd' is referenced in the type of the 'd' argument. Without the '#' this would be an error.

may be {} is universal contifier? https://groups.google.com/forum/#!topic/ats-lang-users/VKuV0kFysxc

https://groups.google.com/forum/#!topic/ats-lang-users/YDS58Hbs2aw